Artificial + Intelligence?

- **AI Concepts**: What are we actually talking about?
  - Clarify what the (modern) research field of AI does, and does not, try to do.
- **AI History**: How did this come about?
  - Just a little background to illustrate how we came from 'classical AI' to 'modern AI'.
- **AI Today**: What is the landscape of techniques and applications?
  - Rough overview, and some examples.

What is Intelligence?

Quiz! can choose multiple answers

- Ability to think . . . ?
- Simulating the brain . . . ?
- Creativity . . . ?
- Ability to learn . . . ?
- Being good at maths . . . ?
- Playing good Chess . . . ?

What is *Artificial* Intelligence?

An engineering standpoint

The science of making machines do things that would require intelligence if done by humans. (M. Minsky)

Is this an operational definition? Hmm...

- How do we know what human activities require intelligence?
- BTW, what is human intelligence?

Acting Humanly with intelligence: Turing Test

- Not reproducible... only a proof of concept?

What is *Artificial* Intelligence?

Another perspective please

The exciting new effort to make computers think. Machines with minds, in the full and literal sense. (J. Haugeland)

Same problems as with Minsky's definition:

- what is thinking?
- what is mind?

What is *Artificial* Intelligence?

A Rational perspective

The branch of computer science that is concerned with the automation of intelligent behavior. (Luger and Stubblefield)

- **Intelligent behavior**: make 'good' (rational) action choices
- Are humans 'rational' agents?

What is *Artificial* Intelligence?

A Rational perspective

The branch of computer science that is concerned with the automation of intelligent behavior. (Luger and Stubblefield)

- **Intelligent behavior**: make 'good' (rational) action choices
- Are humans 'rational' agents? We often make mistakes; we are not all chess grandmasters even though we may know all the rules of chess. More about human systematic errors (*Thinking, fast and slow - Kahneman*)"

Rational Agents

Agents:

- Perceive the environment through sensors (→percepts).
- Act upon the environment through actuators (→actions).

→ Examples? Humans, animals, robots, software agents (softbots), . . .

Rational Agents . . . do 'the right thing'!

→ Any idea what that means, 'do the right thing'?

Rational Agents

Agents:

- Perceive the environment through sensors (→percepts).
- Act upon the environment through actuators (→actions).

→ Examples? Humans, animals, robots, software agents (softbots), . . .

Rational Agents . . . do 'the right thing'!

→ Any idea what that means, 'do the right thing'? Rational agents select their actions so as to maximize a performance measure.

→ Q: What's the performance measure of an autonomous vacuum cleaner?

Rational Agents

Agents:

- Perceive the environment through sensors (→percepts).
- Act upon the environment through actuators (→actions).

→ Examples? Humans, animals, robots, software agents (softbots), . . .

Rational Agents . . . do 'the right thing'!

→ Any idea what that means, 'do the right thing'? Rational agents select their actions so as to maximize a performance measure.

→ Q: What's the performance measure of an autonomous vacuum cleaner? m^2 per hour, Level of cleanliness, Energy usage, . . .

→ What if the vacuum cleaner's sensors are not good enough? click: Robot Revolution news "

Rational Agents

. . . **TRY to do 'the right thing'!**

→ The hypothetical best case ('the right thing') is often unattainable.

→ The agent might not be able to perceive all relevant information. (Is there dirt under this bed?)

Rationality vs. Omniscience:

- An omniscient agent knows everything about the enviroment, and knows the actual effects of its actions.
- A rational agent just makes the best of what it has at its disposal, maximizing expected performance given its percepts and knowledge.

→ Example? I check the traffic before crossing the street. As I cross, I am hit by a meteorite. Was I lacking rationality?

Mapping your input to the best possible output:

Performance measure × Percepts × Knowledge → Action

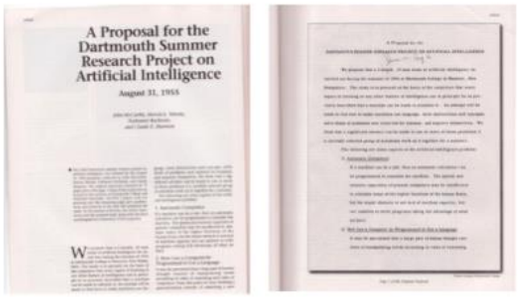
What Does AI Do?

→ Artificial intelligence as an idea can be roughly classified along the dimensions thinking vs. acting and humanly vs. rationally.

	Humanly	Rationally
Thinking	Systems that think like humans (Cognitive Science)	Systems that think rationally (Logics: Knowledge and Deduction)
Acting	Systems that act like humans (Turing Test)	Systems that act rationally (How to make good action choices)

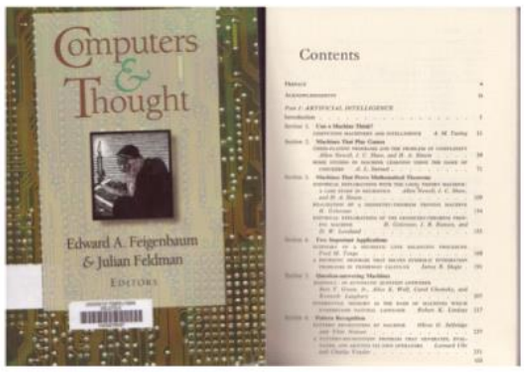
⇝ A central aspect of intelligence (and one possible way to define it) is the ability to act successfully in the world

Origins of AI: Darmouth 1956



The proposal (for the meeting) is to proceed on the basis of the conjecture that every aspect of . . . intelligence can in principle be so precisely described that a machine can be made to simulate it

Computers and Thought 1963



An early collection of AI papers and programs for playing chess and checkers, proving theorems in logic and geometry, planning, etc.

AI: 60's, 70's, and 80's

Many of the key AI contributions in 60's, 70's, and early 80's had to to with programming and the representation of knowledge in programs:

- Lisp (Functional Programming)
- Prolog (Logic Programming)
- Rule-based Programming
- 'Expert Systems' Shells and Architectures

AI Methodology: Theories as Programs

For writing an AI dissertation in the 60's, 70's and 80's, it was common to:

- pick up a task and domain X
- analyze/introspect/find out how task is solved
- capture this reasoning in a program

The dissertation was then

- a theory about X (scientific discovery, circuit analysis, computational humor, story understanding, etc), and
- a program implementing the theory, tested over a few examples.

Many great ideas came out of this work . . . but there was a problem . . .

Methodology Problem

→ Theories expressed as programs cannot be proved wrong: when a program fails, it can always be blamed on 'missing knowledge'

Three approaches to this problem

- narrow the domain (expert systems)
  - problem: lack of generality
- accept the program is just an illustration, a demo
  - problem: limited scientific value
- fill up the missing knowledge (intuition, commonsense)
  - problem: not successful so far

AI Winter: the 80's

→ The knowledge-based approach reached an **impasse** in the 80's, a time also of debates and controversies:

- **Good Old Fashioned AI** is 'rule application' but intelligence is not (Haugeland)

Many criticisms of mainstream AI partially valid then; less valid now.

AI 90's - 2020

Formalization of AI techniques and increased use of mathematics. Recent issues of AIJ, JAIR, AAAI or IJCAI shows papers on:

1. SAT and Constraints
2. Search and Planning
3. Probabilistic Reasoning
4. Probabilistic Planning
5. Inference in First-Order Logic
6. Machine Learning
7. Natural Language
8. Vision and Robotics
9. Multi-Agent Systems

→ Areas 1 to 4 often deemed about techniques, but more accurate to regard them as models and solvers.

Solver Example

Problem ⟹ [*Solver*] ⟹ *Solution*

Example:

- **Problem:** The age of John is 3 times the age of Peter. In 10 years, it will be only 2 times. How old are John and Peter?
- **Expressed as:** J = 3P ; J + 10 = 2(P + 10)
- **Solver:** Gauss-Jordan (Variable Elimination)
- **Solution:** P = 10 ; J = 30

Solver is **general** as deals with any problem expressed as an instance of **model** Linear Equations Model, however, is **tractable**, AI models are not . . .

AI Solver

Problem ⟹ [*Solver*] ⟹ *Solution*

- The basic models and tasks include
  - **Constraint Satisfaction/SAT:** find state that satisfies constraints
  - **Planning Problems:** find action sequence that produces desired state
  - **Planning with Feedback:** find strategy for producing desired state
- Solvers for these models are **general**; not tailored to specific instances
- All of these models are **intractable**, and some extremely powerful (POMDPs)
- The challenge is mainly computational: **how to scale up**
- For this, solvers must **recognize and exploit structure** of the problems
- Methodology is **empirical**: benchmarks and competitions

SAT and CSPs

- **SAT**: determine if there is a **truth assignment** that satisfies a set of clauses

$$x \vee \neg y \vee z \vee \neg w \vee \ldots \quad (1)$$

- Problem is NP-Complete, which in practice means worst-case behavior of SAT algorithms is **exponential** in number of variables ($2^{100} = 10^{30}$)
- Yet current SAT solvers manage to solve problems with **thousands of variables** and clauses, and used widely (circuit design, verification, planning, etc)
- **Constraint Satisfaction Problems (CSPs)** generalize SAT by accommodating non-boolean variables as well, and constraints that are not clauses
- Key is **efficient (poly-time) inference** in every node of search tree: **unit resolution, conflict-based learning**, ...
- Many other ideas **logically possible**, but **do not work** (don't scale up): pure search, pure inference, etc.

Classical Planning Model

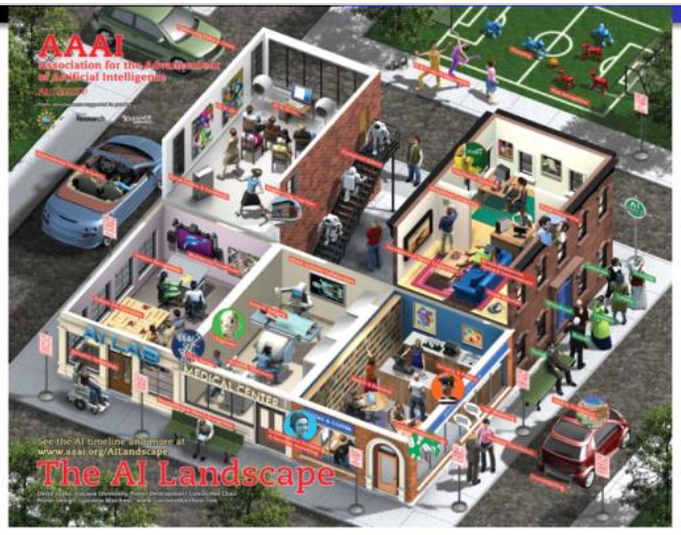
- Planning is the **model-based approach** to autonomous behavior,
- A system can be in one of many **states**
- States assign **values** to a set of **variables**
- **Actions** change the values of certain variables
- **Basic task**: find **action sequence** to drive **initial** state into **goal** state

$$\textit{Model} \Longrightarrow \boxed{\textit{Planner}} \Longrightarrow \textit{Action Sequence}$$

- **Complexity**: NP-hard; i.e., exponential in number of vars in **worst case**
- **Planner** is generic; it should work on any domain no matter what variables are about

Why do we need such an AI?

- **Cheess**: 2 player zero-sum game
- **Music/Speech Recognition**
- **Recommender systems**
- **Medical Diagnosis**: decission support systems
- **Self-driven car**
- **Playing Atari Games** Deep Learning
- ...



Why do we need such AI Planning?

Settings where greater autonomy required:

- **Space Exploration**: (RAX) first artificial intelligence control system to control a spacecraft without human supervision (1998)
- **Business Process Management**
- **First Person Shooters & Games:** classical planners playing Atari Games
- **Interactive Storytelling**
- **Network Security**
- **Logistics/Transportation/Manufacturing**: Multi-model Transportation, forest fire fighting, PARC printer
- **Wherehouse Automation**: Multi-Agent Path Finding, Post China, Amazon
- Automation of Industrial Operations (Schlumberger)
- Self Driving Cars ...

Find out more at ICAPS in Action (right panel)

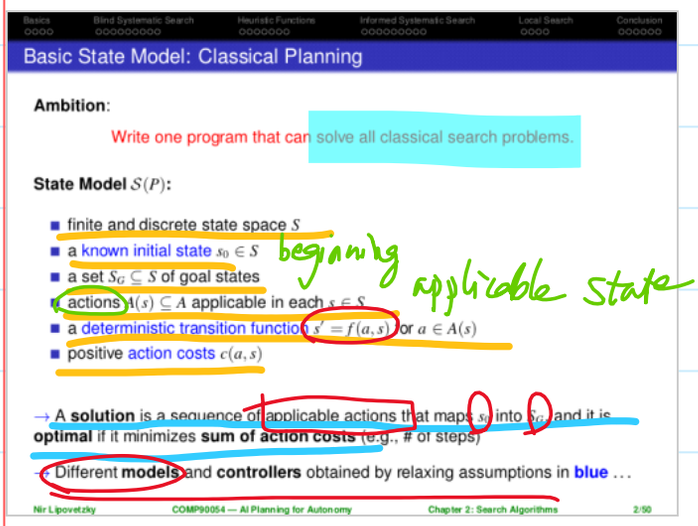
Summary: AI and Automated Problem Solving

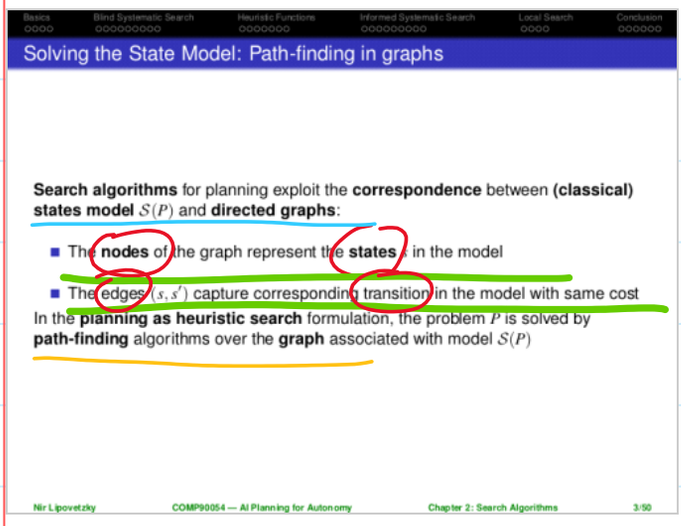
- A **research agenda** that has emerged in last 20 years: **solvers** for a range of **intractable models**
- **Solvers** unlike other programs are **general** as they do not target individual problems but families of problems (**models**)
- The challenge is **computational**: how to scale up
- Sheer **size of problem** shouldn't be impediment to meaningful solution
- **Structure** of given problem must recognized and **exploited**
- Lots of room for **ideas** but methodology **empirical**
- Consistent **progress**
  - effective inference methods (derivation of h, conflict-learning)
  - islands of tractability (treewidth methods and relaxations)
  - transformations (compiling away incomplete info, extended goals, . . .)

Chapter 2

2020年8月3日 星期一 17:28

intro02-pr
e-handout





Classification of Search Algorithms

Blind search vs. heuristic (or informed) search:

- Blind search algorithms: Only use the basic ingredients for general search algorithms.
  - *e.g., Depth First Search (DFS), Breadth-first search (BrFS), Uniform Cost (Dijkstra), Iterative Deepening (ID)*
- Heuristic search algorithms: Additionally use heuristic functions which estimate the distance (or remaining cost) to the goal.
  - *e.g., A*, IDA*, Hill Climbing, Best First, WA*, DFS B&B, LRTA*, ...*

Systematic search vs. local search:

- Systematic search algorithms: Consider a large number of search nodes simultaneously.
- Local search algorithms: Work with one (or a few) candidate solutions (search nodes) at a time.

→ This is not a black-and-white distinction; there are *crossbreeds* (e.g., enforced hill-climbing).

What works where in planning?

Blind search vs. heuristic search:

- For satisficing planning, heuristic search vastly outperforms blind algorithms pretty much everywhwere.
- For optimal planning, heuristic search also is better (but the difference is less pronounced).

Systematic search vs. local search:

- For satisficing planning, there are successful instances of each.
- For optimal planning, systematic algorithms are required.

→ Here, we cover the subset of search algorithms most successful in planning. Only some Blind search algorithms are covered. (refer to Russel & Norvig Chapters 3 and 4 for that).

Search Terminology

Search node n: Contains a *state* reached by the search, plus information about how it was reached.

Path cost $g(n)$: The cost of the path reaching n.

Optimal cost g^*: The cost of an optimal solution path. For a state s, $g^*(s)$ is the cost of a cheapest path reaching s.

Node expansion: Generating all successors of a node, by applying all actions applicable to the node's state s. Afterwards, the *state* s itself is also said to be expanded.

Search strategy: Method for deciding which node is expanded next.

Open list: Set of all *nodes* that currently are candidates for expansion. Also called frontier.

Closed list: Set of all *states* that were already expanded. Used only in graph search, not in tree search (up next). Also called explored set.

World States vs. Search States

Reminder: Search Space for Classical Search

A (classical) search space is defined by the following three operations:

- start(): Generate the start (search) state.
- is-target(s): Test whether a given search state is a target state.
- succ(s): Generates the successor states (a, s') of search state s, along with the actions through which they are reached.

Search states $\neq$ world states:

- Progression: ✓ → forward
- Regression: ✗ → backward

→ We consider progression in the entire course, unless explicitly stated otherwise.
We use "s" to denote world/search states interchangeably.

search states = world states
search states = set of world states

Search States vs. Search Nodes

- **Search states** s: States (vertices) of the search space.
- **Search nodes** σ: Search states, plus information on where/when/how they are encountered during search.

What is in a search node?

Different search algorithms store different information in a search node σ, but typical information includes:

- *state*(σ): Associated search state.
- *parent*(σ): Pointer to search node from which σ is reached.
- *action*(σ): An action leading from *state*(*parent*(σ)) to *state*(σ).
- $g(\sigma)$: Cost of σ (cost of path from the root node to σ).

For the root node, *parent*(σ) and *action*(σ) are undefined.

Criteria for Evaluating Search Strategies

Guarantees:

Completeness: Is the strategy guaranteed to find a solution when there is one?

Optimality: Are the returned solutions guaranteed to be optimal?

Complexity:

Time Complexity: How long does it take to find a solution? (Measured in generated states.)

Space Complexity: How much memory does the search require? (Measured in states.)

Typical state space features governing complexity:

Branching factor b: How many successors does each state have?

Goal depth d: The number of actions required to reach the shallowest goal state.

Before We Begin

No additional work for programmer
Same expansion order regardless of what the problem actually is

Blind search vs. informed search:

- Blind search does not require any input beyond the problem.
  → Pros and Cons?

- Informed search requires as additional input a heuristic function h (Next Chapter) that maps states to estimates of their goal distance.
  → Pros and Cons?

  → Note: In planning, h is generated automatically from the declarative problem description

More effective in practice
need to implement h

→ Note: In planning, h is generated automatically from the declarative problem description

More effective in practice need to implement h

Before We Begin, ctd.

Blind search strategies we'll discuss:

- Breadth-first search. Advantage: time complexity.
  variant: Uniform cost search.
- Depth-first search. Advantage: space complexity.
- Iterative deepening search. Combines advantages of breadth-first search and depth-first search. Uses depth-limited search as a sub-procedure.

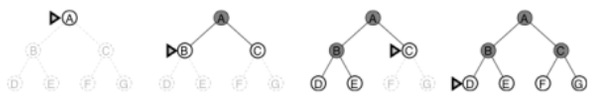
Blind search strategy we won't discuss:

- Bi-directional search. Two separate search spaces, one forward from the initial state, the other backward from the goal. Stops when the two search spaces overlap.

Breadth-First Search: Illustration and Guarantees

Strategy: Expand nodes in the order they were produced (FIFO frontier)

Illustration:



Guarantees:

- Completeness? ✓
- Optimality?

optimal for uniform-cost only

Breadth-First Search: Complexity

Time Complexity: Say that b is the maximal branching factor, and d is the goal depth (depth of shallowest goal state).

- Upper bound on the number of generated nodes?

worst-case

$b + b^2 + \dots + b^d$

$O(b^d)$

- So the time complexity is
- And if we were to apply the goal test at node-expansion time, rather than node-generation time?

$O(b^{d+1})$

Space Complexity:

Breadth-First Search: Example Data

Setting: $b = 10$; 10000 nodes/second; 1000 bytes/node.

Yields data: (inserting values into previous equations)

Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11110	11	milliseconds	10.6	megabytes
6	10^6	1.1	seconds	1	gigabyte
8	10^8	2	minutes	103	gigabytes
10	10^{10}	3	hours	10	terabytes
12	10^{12}	13	days	1	petabyte
14	10^{14}	3.5	years	99	petabytes

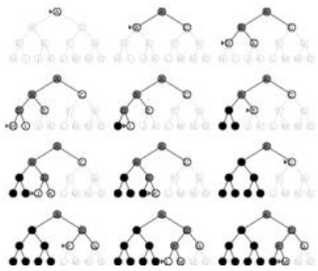
→ So, which is the worse problem, time or memory?

memory

Depth-First Search: Illustration

Strategy: Expand the most recent nodes in (LIFO frontier).

Illustration: (Nodes at depth 3 are assumed to have no successors)



Depth-First Search: Guarantees and Complexity

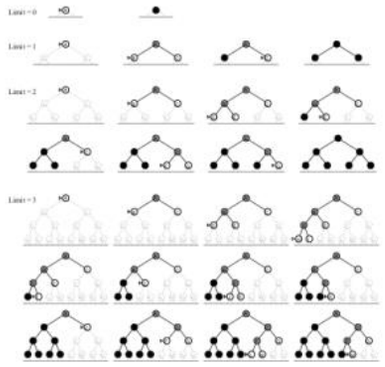
Guarantees:

- Optimality? ✗ hope for the best
- Completeness? ✗ search space is not finite

Complexity: O(bm)

- Space: Stores nodes and applicable actions on the path to the current node. So if m is the maximal depth reached, the complexity is
- Time: If there are paths of length m in the state space, $O(b^m)$ nodes can be generated. Even if there are solutions of depth 1!
  → If we happen to choose "the right direction" then we can find a length-l solution in time $O(b\,l)$ regardless how big the state space is.

Iterative Deepening Search: Illustration



mixed BFS DFS

Iterative Deepening Search: Guarantees and Complexity

"Iterative Deepening Search=
Keep doing the same work over again until you find a solution."

BUT: Optimality? Completeness? ✓ Space complexity? $O(bd)$

uniform cost

Time complexity:

Breadth-First-Search	$b + b^2 + \cdots + b^{d-1} + b^d \in O(b^d)$
Iterative Deepening Search	$(d)b + (d-1)b^2 + \cdots + 3b^{d-2} + 2b^{d-1} + 1b^d \in O(b^d)$

Example: $b = 10, d = 5$

Breadth-First Search	$10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
Iterative Deepening Search	$50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$

→ IDS combines the advantages of breadth-first and depth-first search. It is the preferred blind search method in large state spaces with unknown solution depth.

Heuristic Search Algorithms: Systematic

→ Heuristic search algorithms are the most common and overall most successful algorithms for classical planning.

Systematic heuristic search algorithms:

- Greedy best-first search.
  → One of 3 most popular algorithms in satisficing planning.
- Weighted A*.
  → One of 3 most popular algorithms in satisficing planning.
- A*.
  → Most popular algorithm in optimal planning. (Rarely ever used for satisficing planning.)
- IDA*, depth-first branch-and-bound search, breadth-first heuristic search, ...

The origins: Shakey Video

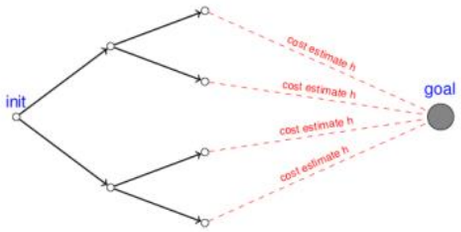
Heuristic Search Algorithms: Local

→ Heuristic search algorithms are the most common and overall most successful algorithms for classical planning.

Local heuristic search algorithms:

- Hill-climbing.
- Enforced hill-climbing.
  → One of 3 most popular algorithms in satisficing planning.
- Beam search, tabu search, genetic algorithms, simulated annealing, ...

Heuristic Search: Basic Idea



→ Heuristic function h estimates the cost of an optimal path to the goal; search gives a preference to explore states with small h.

Heuristic Functions

Heuristic searches require a heuristic function to estimate remaining cost:

Definition (Heuristic Function). *Let Π be a planning task with state space Θ_Π. A heuristic function, short heuristic, for Π is a function $h : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$. Its value $h(s)$ for a state s is referred to as the state's heuristic value, or h-value.*

Definition (Remaining Cost, h^*). *Let Π be a planning task with state space Θ_Π. For a state $s \in S$, the state's remaining cost is the cost of an optimal plan for s, or ∞ if there exists no plan for s. The perfect heuristic for Π, written h^*, assigns every $s \in S$ its remaining cost as the heuristic value.*

h^* perfect heuristic

Heuristic Functions: Discussion

What does it mean to "estimate remaining cost"?

- For many heuristic search algorithms, h does not need to have any properties for the algorithm to "work" (= be correct and complete).
  → h is *any* function from states to numbers ...
- Search performance depends crucially on "how well h reflects h^*"!!
  → This is informally called the informedness or quality of h.
- For some search algorithms, like A*, we can *prove* relationships between formal quality properties of h and search efficiency (mainly the number of expanded nodes).
- For other search algorithms, "it works well in practice" is often as good an analysis as one gets.

→ We will analyze in detail approximations to one particularly important heuristic function in planning: h^+.

Heuristic Functions: Discussion, ctd.

"Search performance depends crucially on the informedness of h ..."
Any other property of h that search performance crucially depends on?

extreme
$h=h^*$ perfectly informed computation expensive
$h=0$ no information computed in constant time

Properties of Heuristic Functions

Definition (Safe/Goal-Aware/Admissible/Consistent). *Let Π be a planning task with state space $\Theta_\Pi = (S, L, c, T, I, S^G)$, and let h be a heuristic for Π. The heuristic is called:*

- *safe* if $h^*(s) = \infty$ *for all* $s \in S$ *with* $h(s) = \infty$;
- *goal-aware* if $h(s) = 0$ *for all goal states* $s \in S^G$;
- *admissible* if $h(s) \leq h^*(s)$ *for all* $s \in s$;
- *consistent* if $h(s) \leq h(s') + c(a)$ *for all transitions* $s \xrightarrow{a} s'$.

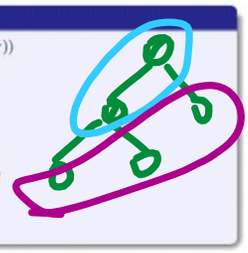
→ **Relationships?**

consistent & goal-aware ⇒ admissible
admissible ⇒ goal-aware
safe
no solution goal state
cost
$h(s) - h(s') \leq c(a)$
lower bound of h^*

Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

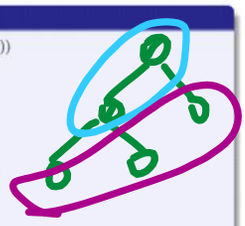
```
open := new priority queue ordered by ascending h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ := open.pop-min() /* get best state */
    if state(σ) ∉ closed: /* check duplicates */
        closed := closed ∪ {state(σ)} /* close state */
        if is-goal(state(σ)): return extract-solution(σ)
        for each (a, s') ∈ succ(state(σ)): /* expand state */
            σ' := make-node(σ, a, s')
            if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```



Greedy Best-First Search

Greedy Best-First Search (with duplicate detection)

```
open := new priority queue ordered by ascending h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    σ := open.pop-min() /* get best state */
    if state(σ) ∉ closed: /* check duplicates */
        closed := closed ∪ {state(σ)} /* close state */
        if is-goal(state(σ)): return extract-solution(σ)
        for each (a, s') ∈ succ(state(σ)): /* expand state */
            σ' := make-node(σ, a, s')
            if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```



Greedy Best-First Search: Remarks

Properties:

- Complete?
- Optimal?
- Invariant under all strictly monotonic transformations of h (e.g., scaling with a positive constant or adding a constant).

Implementation:

- Priority queue: e.g., a min heap.
- "Check Duplicates": Could already do in "expand state"; done here after "get best state" *only* to more clearly point out relation to A*.

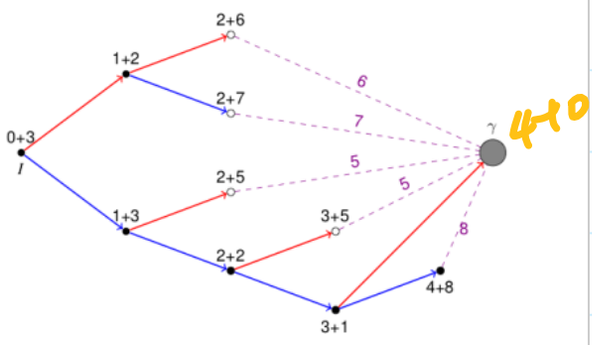
A*

A* (with duplicate detection and re-opening)

```
open := new priority queue ordered by ascending g(state(σ)) + h(state(σ))
open.insert(make-root-node(init()))
closed := ∅
best-g := ∅/* maps states to numbers */
while not open.empty():
    σ := open.pop-min()
    if state(σ) ∉ closed or g(σ) < best-g(state(σ)):
        /* re-open if better g; note that all σ' with same state but worse g
           are behind σ in open, and will be skipped when their turn comes */
        closed := closed ∪ {state(σ)}
        best-g(state(σ)) := g(σ)
        if is-goal(state(σ)): return extract-solution(σ)
        for each (a, s') ∈ succ(state(σ)):
            σ' := make-node(σ, a, s')
            if h(state(σ')) < ∞: open.insert(σ')
return unsolvable
```

A*: Example

g+h



A*: Terminology

- f-value of a state: defined by $f(s) := g(s) + h(s)$.
- Generated nodes: Nodes inserted into *open* at some point.
- Expanded nodes: Nodes σ popped from *open* for which the test against *closed* and *distance* succeeds.
- Re-expanded nodes: Expanded nodes for which *state*$(\sigma) \in$ *closed* upon expansion (also called re-opened nodes).

A*: Remarks

Properties:

- Complete?
- Optimal?

Implementation:

- Popular method: break ties ($f(s) = f(s')$) by smaller h-value.
- If h is admissible and consistent, then A* never re-opens a state. So if we know that this is the case, then we can simplify the algorithm.
- Common, hard to spot bug: check duplicates at the wrong point. (Russel & Norvig are way too imprecise about this.)
- Our implementation is optimized for readability not for efficiency!

Quizz!

Weighted A*

weight

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

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

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 search

g + w·h

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.

Hill-Climbing

Hill-Climbing

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

consider all successors

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this complete or optimal?
- Can easily get stuck in local minima where immediate improvements of $h(\sigma)$ are not possible.
- Many variations: tie-breaking strategies, restarts, ...

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():
    σ = *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)):
        σ' := make-node(σ, a, s')
        queue.push-back(σ')
  fail

$h(s') < h(s_0)$

BFS

↝ Breadth-first search for state with strictly smaller h-value.

$s_0 \rightarrow$ find s that $h(s) < h(s_0)$

Enforced Hill-Climbing, ctd.

Enforced Hill-Climbing

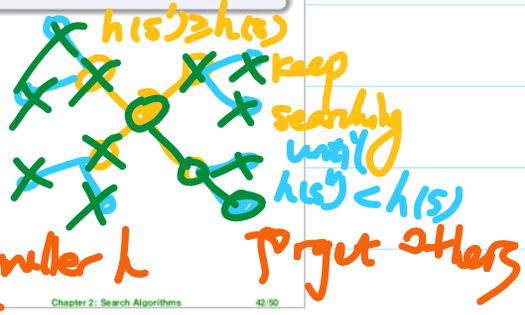
σ := make-root-node(init())
while not is-goal(state(σ)):
  σ := improve(σ)
return extract-solution(σ)

Remarks:

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

complete if h is goal-aware

→ no state with smaller h is reachable from s



Properties of Search Algorithms

	DFS	BrFS	ID	A*	HC	IDA*
Complete	No	Yes	Yes	Yes	No	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes
Time	∞	b^d	b^d	b^d	∞	b^d
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$

- Parameters: d is solution depth; b is branching factor
- Breadth First Search (BrFS) optimal when costs are uniform
- A*/IDA* optimal when h is **admissible**; $h \leq h^*$

with $f(s) = g(s) + h(s)$

$f(s') > t$ → update next t

$t_{\text{next}} = \min_{s'} f(s')$

Properties of Search Algorithms

	DFS	BrFS	ID	A*	HC	IDA*
Complete	No	Yes	Yes	Yes	No	Yes
Optimal	No	Yes*	Yes	Yes	No	Yes
Time	∞	b^d	b^d	b^d	∞	b^d
Space	$b \cdot d$	b^d	$b \cdot d$	b^d	b	$b \cdot d$

- Parameters: d is solution depth; b is branching factor
- Breadth First Search (BrFS) optimal when costs are uniform
- A*/IDA* optimal when h is **admissible**; $h \leq h^*$

Quiz!

Question!

If we set $h(n) := 0$ for all n, what does A* become?

(A): Breadth-first search. (B): Depth-first search.
(C): Uniform-cost search. (D): Depth-limited search.

Question!

If we set $h(n) := 0$ for all n, what can greedy best-first search become?

(A): Breadth-first search. (B): Depth-first search.
(C): Uniform-cost search. (D): A), B) and C)

Quiz!, ctd.

Question!

Is informed search always better than blind search?

(A): Yes. (B): No.

Summary

Distinguish: World states, search states, search nodes.

- World state: Situation in the world modelled by the planning task.
- Search state: Subproblem remaining to be solved.
  - In progression, world states and search states are identical.
  - In regression, search states are sub-goals describing sets of world states.
- Search node: Search state + info on "how we got there".

Search algorithms mainly differ in order of node expansion:

- Blind vs. heuristic (or informed) search.
- Systematic vs. local search.

Summary (ctd.)

- Search strategies differ (amongst others) in the order in which they expand search nodes, and in the way they use duplicate elimination. Criteria for evaluating them are completeness, optimality, time complexity, and space complexity.
- Breadth-first search is optimal but uses exponential space; depth-first search uses linear space but is not optimal. Iterative deepening search combines the virtues of both.

Summary (ctd.)

Heuristic Functions: Estimators for remaining cost.

- Usually: The more informed, the better performance.
- Desiderata: Safe, goal-aware, admissible, consistent.
- The ideal: Perfect heuristic h^*.

Heuristic Search Algorithms:

- Most common algorithms for satisficing planning:
  - Greedy best-first search.
  - Weighted A*.
  - Enforced hill-climbing.
- Most common algorithm for optimal planning:
  - A*.

Reading

- *Artificial Intelligence: A Modern Approach (Third Edition)* , Chapter 3 “Solving Problems by Searching” and the first half of Chapter 4 “Beyond Classical Search”.

  Content: An overview of various search algorithms, including blind searches as well as greedy best-first search and A*.
- Search Tutorial in the context of path-finding `http://www.redblobgames.com/pathfinding/a-star/introduction.html`

Chapter 3

2020年8月20日 星期四 11:01

intro03-pre-handout

COMP90054 — AI Planning for Autonomy

3. Introduction to Planning

How to Describe Arbitrary Search Problems

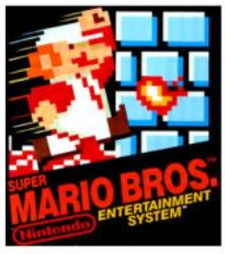
Nir Lipovetzky

THE UNIVERSITY OF MELBOURNE

Semester 2, 2020

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

→ Approaches not orthogonal; successes and limitations in each ...

→ 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

maximize rewards

evolve → iterate

General Problem Solving

Ambition: Write one program that can solve all problems.

→ Write $X \in \{algorithms\}$: for all $Y \in \{`problems'\}$: X'solves'Y

→ 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

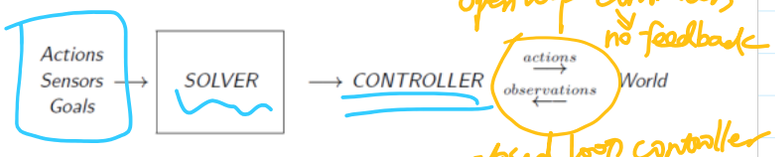
(some new problem) ⇝ (describe problem → use off-the-shelf solver) ⇝ (solution competitive with a human-made specialized program)

→ 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



open loop controllers: no feedback

closed loop controller: have feedback

→ 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

disadvantage: need a model; computationally intractable
- effective loss: Without any domain-specific knowledge about Chess, you don't beat Kasparov

Example: Classical Search Problem

Example: Classical Search Problem

- States: Card positions (position Jspades=Qhearts).
- Actions: Card moves (move Jspades Qhearts freecell4).
- Initial state: Start configuration.
- Goal states: All cards 'home'.
- Solution: Card moves solving this game.

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 $c(a, s)$

→ 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)

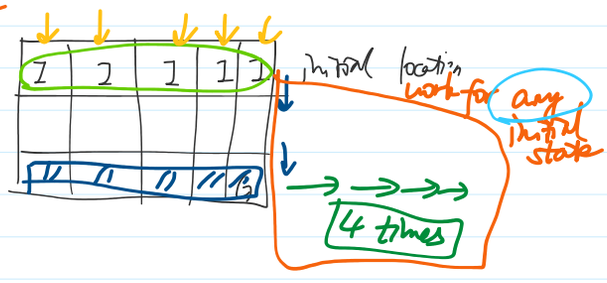
→ Different **models** and **controllers** obtained by relaxing assumptions in **blue** . . .

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 $c(a, s)$

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

→ 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 $c(a, s) > 0$

→ **Solutions** are **functions (policies)** mapping states into actions

→ **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 $P_a(o|s)$, $o \in Obs$

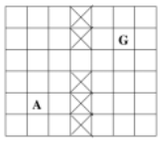
→ **Belief states** are probability distributions over S

→ **Solutions** are policies that map belief states into actions

→ **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

Models, Languages, and Solvers

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

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

language → Map

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

Strips ⊆ PDDL

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) propositions
  - 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 $Add(o) \subseteq F$ → true
  - the Delete list $Del(o) \subseteq F$ → false } Apply
  - the Precondition list $Pre(o) \subseteq F$

which becomes true

which becomes false

when apply

when can you apply

From Language to Models (STRIPS Semantics)

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

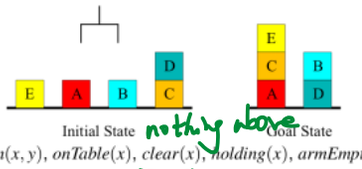
- the states $s \in S$ are collections of atoms from F. $S = 2^F$
- the initial state s_0 is I
- the goal states s are such that $G \subseteq s$
- the actions a in $A(s)$ are ops in O s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

$f(s,a) = s \setminus Del(a) \cup add(a)$

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

→ 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



Initial State Goal State

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

pre: {holding(x), clear(y)}
add: [on(x,y), armEmpty(), clear(x)}
del: {clear(y), holding(x)}

PDDL Quick Facts

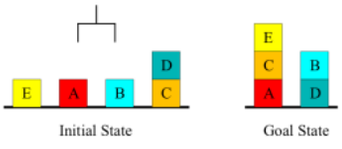
PDDL is not a propositional language:

- Representation is lifted, using object variables to be instantiated from a finite set of objects. (Similar to predicate logic)
- Action schemas parameterized by objects.
- Predicates to be instantiated with objects.

A PDDL planning task comes in two pieces:

- The domain file and the problem file.
- The problem file gives the objects, the initial state, and the goal state.
- The domain file gives the predicates and the operators; each benchmark domain has *one* domain file.

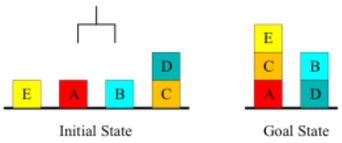
The Blocksworld in PDDL: Domain File



Initial State Goal State

```
(define (domain blocksworld)
(:predicates (clear ?x) (holding ?x) (on ?x ?y)
             (on-table ?x) (arm-empty))
(:action stack
 :parameters (?x ?y)
 :precondition (and (clear ?y) (holding ?x))
 :effect (and (arm-empty) (on ?x ?y)
              (not (clear ?y)) (not (holding ?x)))
)
...
```

The Blocksworld in PDDL: Problem File



Initial State Goal State

```
(define (problem bw-abcde)
(:domain blocksworld)
(:objects a b c d e)
(:init (on-table a) (clear a)
       (on-table b) (clear b)
       (on-table e) (clear e)
       (on-table c) (on d c) (clear d)
       (arm-empty))
(:goal (and (on e c) (on c a) (on b d))))
```


Example: Logistics in Strips PDDL

```
(define (domain logistics)
  (:requirements :strips :typing :equality)
  (:types airport - location  truck airplane - vehicle  vehicle packet - thin
  (:predicates (loc-at ?x - location ?y - city) (at ?x - thing ?y - location)
  (:action load
    :parameters (?x - packet ?y - vehicle ?z - location)
    :precondition (and (at ?x ?z) (at ?y ?z))
    :effect (and (not (at ?x ?z)) (in ?x ?y)))
  (:action unload ..)
  (:action drive
    :parameters (?x - truck ?y - location ?z - location ?c - city)
    :precondition (and (loc-at ?z ?c) (loc-at ?y ?c) (not (= ?z ?y)) (at ?x ?
    :effect (and (not (at ?x ?z)) (at ?x ?y)))
...
(define (problem log3_2)
  (:domain logistics)
  (:objects packet1 packet2 - packet  truck1 truck2 truck3 - truck  airplane1
  (:init (at packet1 office1) (at packet2 office3) ...)
  (:goal (and (at packet1 office2) (at packet2 office2))))
```


Algorithmic Problems in Planning

Satisficing Planning

Input: A planning task P.
Output: A plan for P, or 'unsolvable' if no plan for P exists.

Optimal Planning

Input: A planning task P.
Output: An optimal plan for P, or 'unsolvable' if no plan for P exists.

→ The techniques successful for either one of these are almost disjoint!

→ Satisficing planning is much more effective in practice

→ Programs solving these problems are called (optimal) planners, planning systems, or planning tools.

Decision Problems in Planning

Definition (PlanEx). By PlanEx, we denote the problem of deciding, given a planning task P, whether or not there exists a plan for P.

→ Corresponds to satisficing planning.

Definition (PlanLen). By PlanLen, we denote the problem of deciding, given a planning task P and an integer B, whether or not there exists a plan for P of length at most B.

→ Corresponds to optimal planning.

Reminder (?): NP and PSPACE

Def Turing machine: Works on a tape consisting of tape cells, across which its R/W head moves. The machine has internal states. There are transition rules specifying, given the current cell content and internal state, what the subsequent internal state will be, and whether the R/W head moves left or right or remains where it is. Some internal states are accepting ('yes'; else 'no').

Def NP: Decision problems for which there exists a non-deterministic Turing machine that runs in time polynomial in the size of its input. Accepts if at least one of the possible runs accepts.

Def PSPACE: Decision problems for which there exists a deterministic Turing machine that runs in space polynomial in the size of its input.

Relation: Non-deterministic polynomial space can be simulated in deterministic polynomial space. Thus **PSPACE = NPSPACE**, and hence (trivially) **NP** *subset* **PSPACE**.

→ For comprehensive details, please see a text book. My personal favorite is [Garey and Johnson (1979)].

Computational Complexity of PlanEx and PlanLen

Theorem. PlanEx and PlanLen is **PSPACE**-complete.

→ 'At least as hard as any other problem contained in **PSPACE**.'

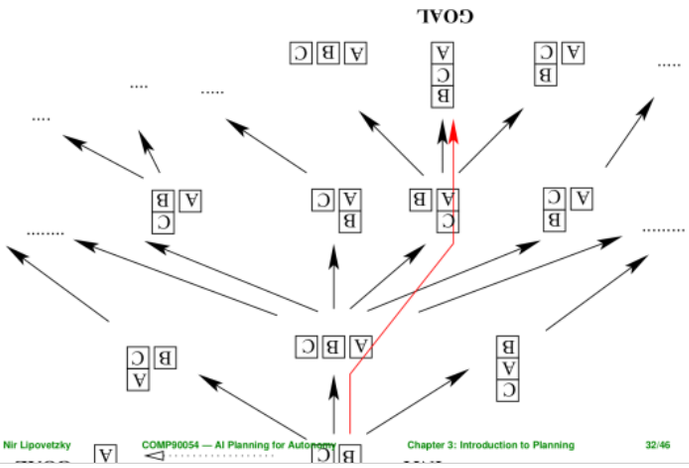
→ Details: [Bylander (1994)]

Domain-Specific PlanEx vs. PlanLen . . .

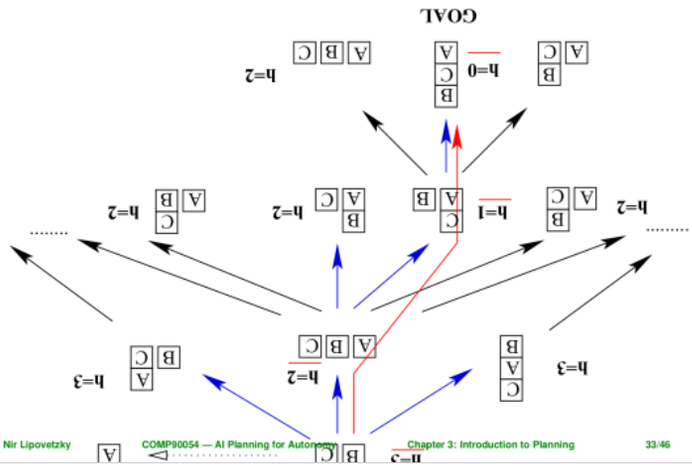
- In general, both have the same complexity.
- Within particular applications, bounded length plan existence is often harder than plan existence.
- This happens in many IPC benchmark domains: PlanLen is **NP**-complete while PlanEx is in **P**.
  - For example: Blocksworld and Logistics.

→ In practice, optimal planning is (almost) never 'easy'

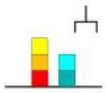
The Blocksworld is Hard?



The Blocksworld is Hard!



So, Why All the Fuss? Example Blocksworld



- n blocks, 1 hand.
- A single action either takes a block with the hand or puts a block we're holding onto some other block/the table.

blocks	states	blocks	states
1	1	9	4596553
2	3	10	58941091
3	13	11	824073141
4	73	12	12470162233
5	501	13	202976401213
6	4051	14	3535017524403
7	37633	15	65573803186921
8	394353	16	1290434218669921

→ State spaces may be huge. In particular, the state space is typically exponentially large in the size of its specification via the problem Π (up next).

→ In other words: Search problems typically are computationallly hard (e.g., optimal Blocksworld solving is **NP**-complete).

Computation: how to solve Strips planning problems?

Key issue: exploit two roles of **language**:

- specifcation: concise model description
- computation: reveal useful heuristic information (structure)

Two traditional approaches: search vs. decomposition

- explicit search of the state model $S(P)$ direct but not effective til recently
- near decomposition of the planning problem thought a better idea

Computational Approaches to Classical Planning

- **General Problem Solver (GPS) and Strips** (50's-70's): mean-ends analysis, decomposition, regression, ...
- **Partial Order (POCL) Planning** (80's): work on any open subgoal, resolve threats; UCPOP 1992
- **Graphplan** (1995 – 2000): build graph containing all possible **parallel** plans up to certain length; then extract plan by searching the graph backward from Goal
- **SATPlan** (1996 – ...): map planning problem given horizon into SAT problem; use state-of-the-art SAT solver
- **Heuristic Search Planning** (1996 – ...): search state space $S(P)$ with heuristic function h extracted from problem P
- **Model Checking Planning** (1998 – ...): search state space $S(P)$ with 'symbolic' Breadth first search where sets of states represented by formulas implemented by BDDs ...

State of the Art in Classical Planning

- significant **progress** since Graphplan
- **empirical methodology**
  - standard PDDL language
  - planners and benchmarks available; competitions
  - focus on performance and scalability
- **large problems solved** (non-optimally)
- different **formulations** and **ideas**
  1. Planning as **Heuristic Search**
  2. Planning as **SAT**
  3. **Other:** Local Search (LPG), Monte-Carlo Search (Arvand), ...

I'll focus on **1** mainly, and partially on **2**

The International Planning Competition (IPC)

Competition?

'Run competing planners on a set of benchmarks devised by the IPC organizers.
Give awards to the most effective planners.'

- 1998, 2000, 2002, 2004, 2006, 2008, 2011, 2014, 2018, 2019
- PDDL [McDermott and others (1998); Fox and Long (2003); Hoffmann and Edelkamp (2005)]
- $\approx$ 40 domains, $\gg$ 1000 instances, 74 (!!) planners in 2011
- Optimal track vs. satisficing track
- Various others: uncertainty, learning, . . .

http://ipc.icaps-conference.org/

... Winners

- IPC 2000: Winner FF, heuristic search (HS), IPC 2002: Winner LPG, HS
- IPC 2004: Winner satisficing SGPlan, HS; optimal SATPLAN, compilation to SAT
- IPC 2006: Winner satisficing SGPlan, HS; optimal SATPLAN, compilation to SAT
- IPC 2008: Winner satisficing LAMA, HS; optimal Gamer, symbolic search
- IPC 2011: Winner satisficing LAMA, HS; optimal Fast-Downward, HS
- IPC 2014: Winner satisficing IBACOP, HS Portfolio; optimal SymbA*, symbolic search
- IPC 2018: Winner satisficing FD/BFWS-LAPKT, HS Portfolio/Width-Based planning; optimal
- IPC 2019: Winner PROBE and BFWS-LAPKT, HS/Width-Based planning

$\rightarrow$ For the rest of this chapter, we focus on planning as heuristic search
$\rightarrow$ This is a VERY short summary of the history of the IPC! There are many different categories, and many different awards.

Disclaimer on IPC

Question

If planners x, y both compete in IPC'YY, and x wins, is x 'better than' y?

(A): Yes. (B): No.

→ Yes, but only on the IPC'YY benchmarks, and only according to the criteria used for determining a 'winner'! On other domains and/or according to other criteria, you may well be better off with the 'looser'.

→ It's complicated, over-simplification is dangerous. (But, of course, nevertheless is being done all the time).

Summary

- General problem solving attempts to develop solvers that perform well across a large class of problems.
- Planning, as considered here, is a form of general problem solving dedicated to the class of classical search problems. (Actually, we also address inaccessible, stochastic, dynamic, continuous, and multi-agent settings.)
- Classical search problems require to find a path of actions leading from an initial state to a goal state.
- They assume a single-agent, fully-observable, deterministic, static environment. Despite this, they are ubiquitous in practice.
- Heuristic search planning has dominated the International Planning Competition (IPC). We focus on it here.
- STRIPS is the simplest possible, while reasonably expressive, language for our purposes. It uses Boolean variables (facts), and defines actions in terms of precondition, add list, and delete list.
- Plan existence (bounded or not) is **PSPACE**-complete to decide for STRIPS.
- PDDL is the de-facto standard language for describing planning problems.

Reading, ctd.

- *Everything You Always Wanted to Know About Planning (But Were Afraid to Ask)* [Joerg Hoffmann, 2011]

  Available at:

  `http://fai.cs.uni-saarland.de/hoffmann/papers/ki11.pdf`

  Content: Joerg personal perspective on planning. Very modern indeed. Excerpt from the abstract:

  The area has long had an affinity towards playful illustrative examples, imprinting it on the mind of many a student as an area concerned with the rearrangement of blocks, and with the order in which to put on socks and shoes (not to mention the disposal of bombs in toilets). Working on the assumption that this "student" is you – the readers in earlier stages of their careers – I herein aim to answer three questions that you surely desired to ask back then already:

  What is it good for? Does it work? Is it interesting to do research in?

Extra material

Introduction to STRIPS, from simple games to StarCraft:
http://www.primaryobjects.com/2015/11/06/artificial-intelligence-planning-with-strips-a-gentle-introduction/

Online/Offline Editor to model in PDDL:
http://editor.planning.domains
https://marketplace.visualstudio.com/items?itemName=jan-dolejsi.pddl&ssr=false#overview

Chapter 4

2020年8月27日 星期四 15:06

intro04-pre-handout

COMP90054 — AI Planning for Autonomy

4. Generating Heuristic Functions

How to Relax: Formally, and Informally, and During Search

Nir Lipovetzky

THE UNIVERSITY OF MELBOURNE

Semester 2, 2020

Agenda

Motivation

→ "Relax"ing is a methodology to construct heuristic functions.

- You can use it when programming a solution to some problem you want/need to solve.
- Planning systems can use it to derive a heuristic function **automatically** from the planning task description (the PDDL input).
  - **Note 1:** If the user had to supply the heuristic function by hand, then we would lose our two main selling points (generality & autonomy & flexibility & rapid prototyping, cf. → Lecture 1-2).
  - **Note 2:** It can of course be of advantage to give the user the *possibility* to (conveniently) supply additional heuristics. Not covered in this course.

How to Relax Informally

How To Relax:

- You have a problem, $\mathcal{P}$, whose perfect heuristic h^* you wish to estimate.
- You define a simpler problem, $\mathcal{P}'$, whose perfect heuristic h'^* can be used to estimate h^*.
- You define a transformation, r, that simplifies instances from $\mathcal{P}$ into instances $\mathcal{P}'$.
- Given $\Pi \in \mathcal{P}$, you estimate $h^*(\Pi)$ by $h'^*(r(\Pi))$.

→ Relaxation means to simplify the problem, and take the solution to the simpler problem as the heuristic estimate for the solution to the actual problem.

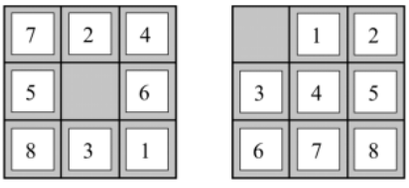
Relaxation in Route-Finding

How to derive straight-line distance by relaxation?

- Problem $\mathcal{P}$: Route finding.
- Simpler problem $\mathcal{P}'$: Route Finding for birds
- Perfect heuristic h'^* for $\mathcal{P}'$: Straight-line distance
- Transformation r: Pretend to be a bird

↓ fully connected graph

Relaxation in the 8-Puzzle



Start State Goal State

Perfect heuristic h^* for $\mathcal{P}$: Actions = "A tile can move from square A to square B if A is adjacent to B and B is blank."

- How to derive the Manhattan distance heuristic? move (A,B) if adjacent (A,B)
- How to derive the misplaced tiles heuristic? move (A,B) without preconds
- h'^* (resp. r) in both: optimal cost in $\mathcal{P}'$ (resp. use different actions).
- Here: Manhattan distance = , misplaced tiles = .

"Goal-Counting" Relaxation in Australia

- Propositions P: $at(x)$ for $x \in \{Sy, Ad, Br, Pe, Da\}$; $v(x)$ for $x \in \{Sy, Ad, Br, Pe, Da\}$.
- Actions $a \in A$: $drive(x, y)$ where x, y have a road; $pre_a = \{at(x)\}$, $add_a = \{at(y), v(y)\}$, $del_a = \{at(x)\}$.
- Initial state I: $at(Sy), v(Sy)$.
- Goal G: $at(Sy), v(x)$ for all x.

Let's "act as if we could achieve each goal directly":

- Problem $\mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $\mathcal{P}'$: All STRIPS planning tasks with empty preconditions and deletes.
- Perfect heuristic h'^* for $\mathcal{P}'$: Optimal plan cost ($= h^+$).
- Transformation $\mathcal{P} \mapsto \mathcal{P}'$: drop preconditions and deletes
- Heuristic value here? 4 .

→ Optimal STRIPS planning with empty preconditions and deletes is still **NP**-hard! (Reduction from MINIMUM COVER, of goal set by add lists.)

→ Need to approximate the perfect heuristic h'^* for $\mathcal{P}'$. Hence goal counting: just approximate h'^* by number-of-false-goals.

How to Relax Formally: Before We Begin

- The definition on the next slide is not to be found in any textbook, and not even in any paper.
- Methods generating heuristic functions differ widely, and it is quite difficult (impossible?) to make *one* definition capturing them *all* in a natural way.
- Nevertheless, a formal definition is useful to state precisely what are the relevant distinction lines in practice.
- The present definition does, I think, do a rather good job of this.
  → It nicely fits what is currently used in planning.
  → It is flexible in the distinction lines, and it captures the basic construction, as well as the essence of all relaxation ideas.

Relaxations

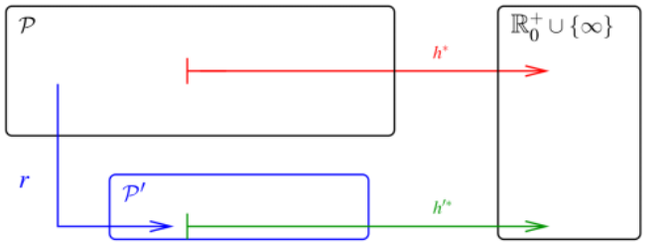
Definition (Relaxation). *Let $h^* : \mathcal{P} \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ be a function. A relaxation of h^* is a triple $\mathcal{R} = (\mathcal{P}', r, h'^*)$ where $\mathcal{P}'$ is an arbitrary set, and $r : \mathcal{P} \mapsto \mathcal{P}'$ and $h'^* : \mathcal{P}' \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ are functions so that, for all $\Pi \in \mathcal{P}$, the relaxation heuristic $h^{\mathcal{R}}(\Pi) := h'^*(r(\Pi))$ satisfies $h'^*(\Pi) \leq h^*(\Pi)$. The relaxation is:*

- *naive if $\mathcal{P}' = \mathcal{P}$ and $h'^* = h^*$;*
- *efficiently constructible if there exists a polynomial-time algorithm that, given $\Pi \in \mathcal{P}$, computes $r(\Pi)$;*
- *efficiently computable if there exists a polynomial-time algorithm that, given $\Pi' \in \mathcal{P}'$, computes $h'^*(\Pi')$.*

Reminder:

- You have a problem, $\mathcal{P}$, whose perfect heuristic h^* you wish to estimate.
- You define a simpler problem, $\mathcal{P}'$, whose perfect heuristic h'^* can be used to (admissibly!) estimate h^*
- You define a transformation, r, from $\mathcal{P}$ into $\mathcal{P}'$.
- Given $\Pi \in \mathcal{P}$, you estimate $h^*(\Pi)$ by $h'^*(r(\Pi))$.

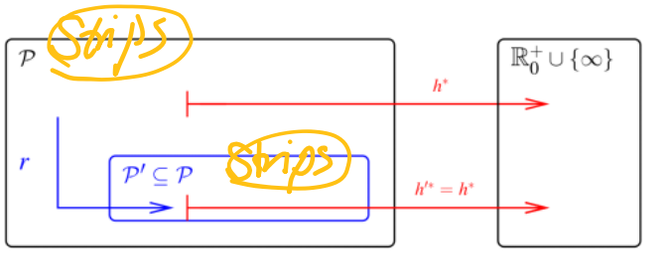
Relaxations: Illustration



Example route-finding:

- Problem $\mathcal{P}$: Route finding.
- Simpler problem $\mathcal{P}'$: Route finding for birds
- Perfect heuristic h'^* for $\mathcal{P}'$: Straight line distance
- Transformation r: Instead you're a bird

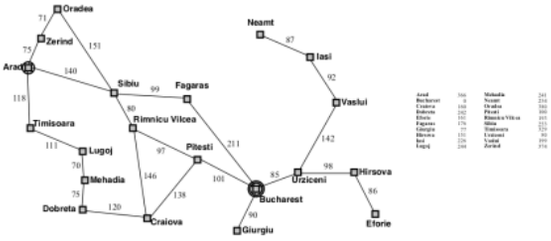
Native Relaxations: Illustration



Example "goal-counting":

- Problem $\mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $\mathcal{P}'$: All STRIPS planning tasks with empty preconditions and deletes.
- Perfect heuristic h'^* for $\mathcal{P}'$: Optimal plan cost $= h^*$.
- Transformation r: drop the preconditions and deletes

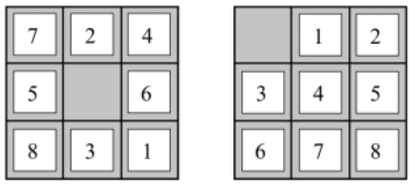
Relaxation in Route-Finding: Properties



Relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$**:** Pretend you're a bird.

- Native? No
- Efficiently constructible? Yes
- Efficiently computable? Yes

Relaxation in the 8-Puzzle: Properties



Start State Goal State

Relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$**:** Use more generous actions rule to obtain Manhattan distance.

- Native? NO
- Efficiently constructible?
- Efficiently computable?

What shall we do with the relaxation?

What if $\mathcal{R}$ is not efficiently constructible?

- Either (a) approximate r, or (b) design r in a way so that it will typically be feasible, or (c) just live with it and hope for the best.
- Vast majority of known relaxations (in planning) are efficiently constructible.

What if $\mathcal{R}$ is not efficiently computable?

- Either (a) approximate h'^*, or (b) design h'^* in a way so that it will typically be feasible, or (c) just live with it and hope for the best.
- Many known relaxations (in planning) are efficiently computable, some aren't. The latter use (a); (b) and (c) are not used anywhere right now.

"Goal-Counting" Relaxation in Australia: Properties

- Propositions P: $at(x)$ for $x \in \{Sy, Ad, Br, Pe, Da\}$; $v(x)$ for $x \in \{Sy, Ad, Br, Pe, Da\}$.
- Actions $a \in A$: $drive(x, y)$ where x, y have a road; $pre_a = \{at(x)\}$, $add_a = \{at(y), v(y)\}$, $del_a = \{at(x)\}$.
- Initial state I: $at(Sy), v(Sy)$.
- Goal G: $at(Sy), v(x)$ for all x.

Relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$**:** Remove preconditions and deletes, then use h^*.

- Native? yes
- Efficiently constructible? yes
- Efficiently computable? No

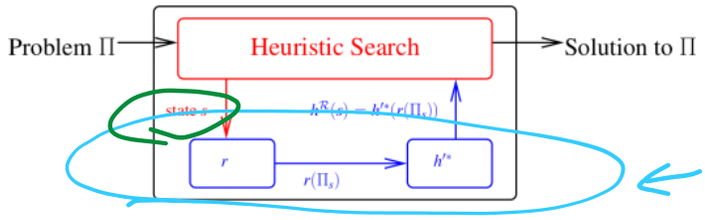
h^*

What shall we do with the relaxation?

goal counting → approximation

How to Relax During Search: Diagram

Using a relaxation $\mathcal{R} = (\mathcal{P}', r, h'^*)$ **during search:**



relax

→ Π_s: Π with initial state replaced by s, i.e., $\Pi = (F, A, c, I, G)$ changed to (F, A, c, s, G).

→ The task of finding a plan for search state s.

→ We will be using this notation in the course!

Questionnaire

Question!

Say we have a robot with one gripper, two rooms A and B, and n balls we must transport. The actions available are $moveXY$, $pickB$ and $dropB$; say $h =$"number of balls not yet in room B". Can h be derived as $h^{\mathcal{R}}$ for a relaxation $\mathcal{R}$?

(A): No.

(B): Yes, just drop the deletes

(C): Sure, *every* admissible h can be derived via a relaxation.

(D): I'd rather relax at the beach.

Summary

- Relaxation is a method to compute heuristic functions.
- Given a problem $\mathcal{P}$ we want to solve, we define a relaxed problem $\mathcal{P}'$. We derive the heuristic by mapping into $\mathcal{P}'$ and taking the solution to this simpler problem as the heuristic estimate.
- Relaxations can be native, efficiently constructible, and/or efficiently computable. None of this is a strict requirement to be useful.
- During search, the relaxation is used *only inside* the computation of the heuristic function on each state; the relaxation does not affect anything else. (This can be a bit confusing especially for native relaxations like ignoring deletes.)

Remarks

The goal-counting approximation h =“count the number of goals currently not true” is **a very uninformative heuristic function:**

- Range of heuristic values is small $(0 \ldots |G|)$.
- We can transform any planning task into an equivalent one where $h(s) = 1$ for all non-goal states s. How?
- Ignores almost all structure: Heuristic value does not depend on the actions at all!

→ By the way, is h safe/goal-aware/admissible/consistent?

→ We will see in → **the next lecture** how to compute **much** better heuristic functions.

Chapter 5

2020年9月3日 星期四 15:29

intro05-pre-handout

COMP90054 — AI Planning for Autonomy

5. Delete Relaxation Heuristics

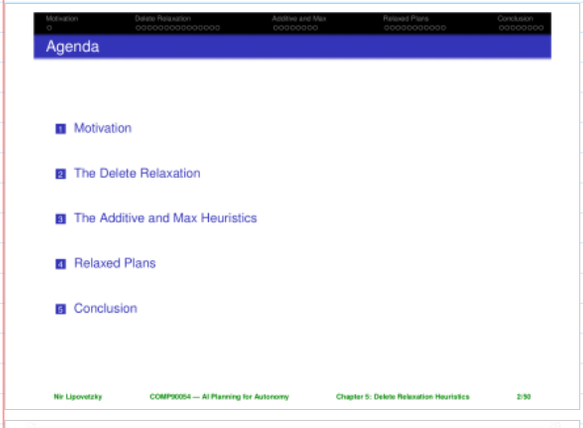
It's a Long Way to the Goal, But How Long Exactly?

Part I: *Acting As If the World Can Only Get Better*

Nir Lipovetzky

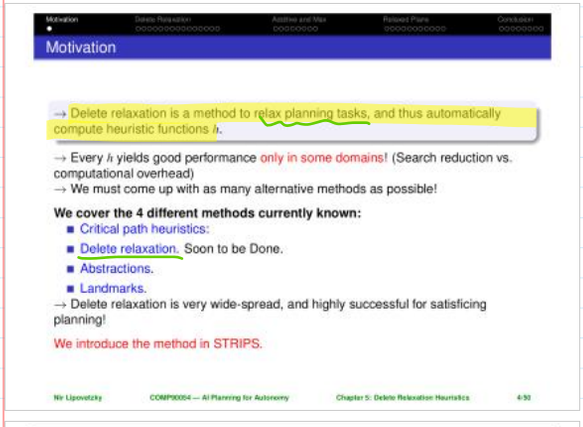
THE UNIVERSITY OF MELBOURNE

Semester 2, 2020



Agenda

1. Motivation
2. The Delete Relaxation
3. The Additive and Max Heuristics
4. Relaxed Plans
5. Conclusion



Motivation

→ Delete relaxation is a method to relax planning tasks, and thus automatically compute heuristic functions h.

→ Every h yields good performance only in some domains! (Search reduction vs. computational overhead)

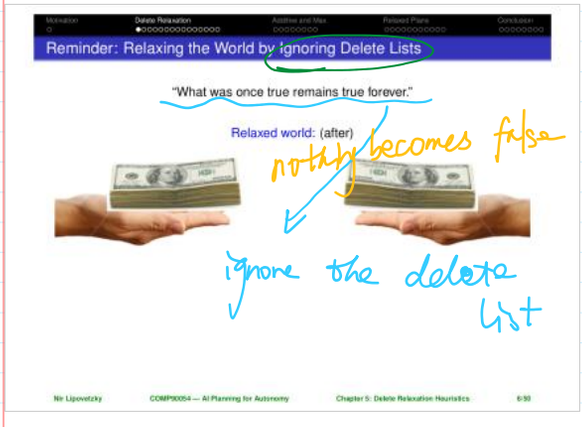
→ We must come up with as many alternative methods as possible!

We cover the 4 different methods currently known:

- Critical path heuristics.
- Delete relaxation. Soon to be Done.
- Abstractions.
- Landmarks.

→ Delete relaxation is very wide-spread, and highly successful for satisficing planning!

We introduce the method in STRIPS.



Reminder: Relaxing the World by Ignoring Delete Lists

"What was once true remains true forever."

Relaxed world: (after)

nothing becomes false

ignore the delete list

The Delete Relaxation

Definition (Delete Relaxation).

1. For a STRIPS action a, by a^+ we denote the corresponding delete relaxed action, or short relaxed action, defined by $pre_{a^+} := pre_a$, $add_{a^+} := add_a$, and $del_{a^+} := \emptyset$.
2. For a set A of STRIPS actions, by A^+ we denote the corresponding set of relaxed actions $A^+ := \{a^+ \mid a \in A\}$; similarly, for a sequence $\vec{a} = \langle a_1, \dots, a_n \rangle$ of STRIPS actions, by $\vec{a}^+$ we denote the corresponding sequence of relaxed actions, $\vec{a}^+ := \langle a_1^+, \dots, a_n^+ \rangle$.
3. For a STRIPS planning task $\Pi = (F, A, c, I, G)$, by $\Pi^+ := (F, A^+, c, I, G)$ we denote the corresponding (delete) relaxed planning task.

→ "+" super-script = delete relaxed. We'll also use this to denote states encountered within the relaxation. (For STRIPS, s^+ is a fact set just like s.)

Definition (Relaxed Plan). Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state. An (optimal) relaxed plan for s is an (optimal) plan for Π_s^+. A relaxed plan for I is also called a relaxed plan for Π.

→ Anybody remember what Π_s is?

A Relaxed Plan for "TSP" in Australia

1. **Initial state:** $\{at(Sy), v(Sy)\}$.
2. **Apply** $drive(Sy, Br)^+$:
3. **Apply** $drive(Sy, Ad)^+$:
4. **Apply** $drive(Ad, Pe)^+$:
5. **Apply** $drive(Ad, Da)^+$:

State Dominance

Definition (Dominance). Let $\Pi^+ = (F, A^+, c, I, G)$ be a STRIPS planning task, and let s^+, s'^+ be states. We say that s'^+ dominates s^+ if $s'^+ \supseteq s^+$.

→ For example, on the previous slide, who dominates who?

Proposition (Dominance). Let $\Pi^+ = (F, A^+, c, I, G)$ be a STRIPS planning task, and let s^+, s'^+ be states where s'^+ dominates s^+. We have:

1. If s^+ is a goal state, then s'^+ is a goal state as well.
2. If $\vec{a}^+$ is applicable in s^+, then $\vec{a}^+$ is applicable in s'^+ as well, and $appl(s'^+, \vec{a}^+)$ dominates $appl(s^+, \vec{a}^+)$.

Proof. (i) is trivial. (ii) by induction over the length n of $\vec{a}^+$. Base case $n = 0$ is trivial. Inductive case $n \to n+1$ follows directly from induction hypothesis and the definition of $appl(.,.)$.

→ It is always better to have more facts true.

The Delete Relaxation and State Dominance

Proposition. Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let $a \in A$. Then $appl(s, a^+)$ dominates both (i) s and (ii) $appl(s, a)$.

Proof. Trivial from the definitions of $appl(s, a)$ and a^+.

⟹ Optimal relaxed plans admissibly estimate the cost of optimal plans:

Proposition (Delete Relaxation is Admissible). Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let $\vec{a}$ be a plan for Π_s. Then $\vec{a}^+$ is a relaxed plan for s.

Proof. Prove by induction over the length of $\vec{a}$ that $appl(s, \vec{a}^+)$ dominates $appl(s, \vec{a})$. Base case is trivial, inductive case follows from (ii) above.

⟹ It is now clear how to find a relaxed plan:

- Applying a relaxed action can only ever make more facts true (i) above).
- That can only be good, i.e., cannot render the task unsolvable (dominance proposition).

→ So?

Greedy Relaxed Planning

Greedy Relaxed Planning for Π_s^+

$s^+ := s$; $\vec{a}^+ := \langle\rangle$
while $G \not\subseteq s^+$ **do**
  if $\exists a \in A$ s.t. $pre_a \subseteq s^+$ and $appl(s^+, a^+) \neq s^+$ **then**
    select one such a
    $s^+ := appl(s^+, a^+)$; $\vec{a}^+ := \vec{a}^+ \circ \langle a^+ \rangle$
  else return "Π_s^+ is unsolvable" **endif**
endwhile
return $\vec{a}^+$

Proposition. *Greedy relaxed planning is sound, complete, and terminates in time polynomial in the size of Π.*

Proof. Soundness: If $\vec{a}^+$ is returned then, by construction, $G \subseteq appl(s, \vec{a}^+)$. Completeness: If "$\Pi_s^+$ is unsolvable" is returned, then no relaxed plan exists for s^+ at that point; since s^+ dominates s, by the dominance proposition this implies that no relaxed plan can exist for s. Termination: Every $a \in A$ can be selected at most once because afterwards $appl(s^+, a^+) = s^+$.

⟹ It is easy to decide whether a relaxed plan exists!

Greedy Relaxed Planning to Generate a Heuristic Function?

Using greedy relaxed planning to generate h

- In search state s during forward search, run greedy relaxed planning on Π_s^+.
- Set $h(s)$ to the cost of $\vec{a}^+$, or ∞ if "Π_s^+ is unsolvable" is returned.

→ Is this heuristic safe? ✓

→ Is this heuristic goal-aware? ✓

→ Is this heuristic admissible? ✗ relaxed plan not optimal

select actions randomly

→ To be informed (accurately estimate h^*), a heuristic needs to approximate the *minimum effort* needed to reach the goal. Greedy relaxed planning doesn't do this because it may select arbitrary actions that aren't relevant at all.

h^+: The Optimal Delete Relaxation Heuristic

Definition (h^+). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task with state space $\Theta_\Pi = (S, A, c, T, I, G)$. The optimal delete relaxation heuristic h^+ for Π is the function $h^+ : S \mapsto \mathbb{R}_0^+ \cup \{\infty\}$ where $h^+(s)$ is defined as the cost of an optimal relaxed plan for s.*

Corollary (h^+ is Admissible). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task. Then h^+ is admissible, and thus safe and goal-aware.* (By admissibility of delete relaxation.)

→ To be informed (accurately estimate h^*), a heuristic needs to approximate the *minimum effort* needed to reach the goal. h^+ naturally does so by asking for the cheapest possible relaxed plans.

[→ You might rightfully ask "But won't optimal relaxed plans usually under-estimate h^*?" Yes, but that's just the effect of considering a relaxed problem, and arbitrarily adding actions useless within the relaxation does not help to address it.]

$h^+ << h^*$

h^+ in "TSP" in Australia

- P: $at(x)$ for $x \in \{Sy, Ad, Br, Pr, Ad'\}$; $v(x)$ for $x \in \{Sy, Ad, Br, Pr, Ad\}$.
- A: $drive(x, y)$ where x, y have a road.

$$c(drive(x,y)) = \begin{cases} 1 & \{x,y\} = \{Sy, Br\} \\ 1.5 & \{x,y\} = \{Sy, Ad\} \\ 3.5 & \{x,y\} = \{Ad, Pr\} \\ 4 & \{x,y\} = \{Ad, Br\} \end{cases}$$

- I: $at(Sy), v(Sy)$; G: $at(Sy), v(x)$ for all x.

$h^* = 20$

Planning vs. Relaxed Planning:

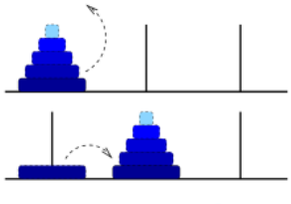
- Optimal plan: drive(Sy, Br), drive(Br, Sy), drive(Sy, Ad), drive(Ad, Pe), drive(Pe, Ad), drive(Ad, Ad'), drive(Ad', Ad), drive(Ad, Sy)
- Optimal relaxed plan: drive(Sy, Br), drive(Sy, Ad), drive(Ad, Pe), drive(Ad, Ad')
- $h^*(I) \geq h^+(I) = 10$

$h^+ = 10$

Reminder: h^+ in (the real) TSP

$h^+(\text{TSP}) = \text{Minimum Spanning Tree!}$

Reminder: h^+ in Hanoi



$h^+(\text{Hanoi}) = n$, not 2^n

But How to Compute h^+?

Definition (Optimal Relaxed Planning). *By PlanOpt$^+$, we denote the problem of deciding, given a STRIPS planning task $\Pi = \langle F, A, c, I, G \rangle$ and $B \in \mathbb{R}_0^+$, whether there exists a relaxed plan for Π whose cost is at most B.*

→ By computing h^+, we would solve PlanOpt$^+$.

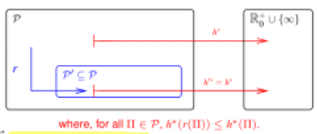
Theorem (Optimal Relaxed Planning is Hard). *PlanOpt$^+$ is NP-complete.*

Proof. Membership:

Hardness: By reduction from SAT.

- For each variable $v_i \in \{v_1, \ldots, v_m\}$ in the CNF, three facts v_i, nov_i, and sev_i; for each clause $c_j \in \{c_1, \ldots, c_n\}$ in the CNF, one fact sat_j.
- Actions $setvtrue_i$: $\langle \emptyset, \{v_i, sev_i\}, \emptyset \rangle$ and $setfalse_i$: $\langle \emptyset, \{nov_i, sev_i\}, \emptyset \rangle$.
- Actions $makesat_{ij}$: $\langle \{v_i\}, \{sat_j\}, \emptyset \rangle$ where v_i appears positively in clause c_j; $\langle \{nov_i\}, \{sat_j\}, \emptyset \rangle$ where v_i appears negatively in clause c_j.
- Initial state $\emptyset$, goal $\{sev_1, \ldots, sev_m, sat_1, \ldots, sat_n\}$; $B := m + n$.

h^+ as a Relaxation Heuristic



where, for all $\Pi \in \mathcal{P}$, $h^*(r(\Pi)) \leq h^*(\Pi)$.

For $h^+ = h^* \circ r$:

- Problem $\mathcal{P}$: All STRIPS planning tasks.
- Simpler problem $\mathcal{P}'$: All STRIPS planning tasks with empty deletes.
- Perfect heuristic $h^{*'}$ for $\mathcal{P}'$: Optimal plan cost = h^* on $\mathcal{P}'$.
- Transformation r: Drop the deletes.

→ Is this a native relaxation?
→ Is this relaxation efficiently constructible?
→ Is this relaxation efficiently computable?

NP-complete

What shall we do with this relaxation?

$h^{max} \leq h^+ \leq h^*$ polynomial

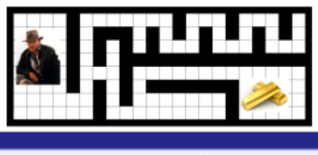
Reminder: → Lecture 4

What if $\mathcal{R}$ is not efficiently computable?

- Either (a) approximate $h^{*'}$, or (b) design h'^* in a way so that it will typically be feasible, or (c) just live with it and hope for the best.
- Many known relaxations (in planning) are efficiently computable, some aren't (like h^+). The latter use (a); (b) and (c) are not used anywhere right now.

→ The delete relaxation heuristic we want is h^+. Unfortunately, this is hard to compute so the computational overhead is very likely to be prohibitive. All implemented systems using the delete relaxation approximate h^+ in one or the other way. We now look at the the most wide-spread approaches to do so.

Quizz@PollEV.com/nirlipovetzk000



Question!

In this domain, h^+ is equal to?

(A): Manhattan Distance. (B): h^*.
(C): Horizontal distance. (D): Vertical distance.

The Additive and Max Heuristics

Definition (h^{add}). *Let $\Pi = \langle F, A, c, I, G \rangle$ be a STRIPS planning task. The additive heuristic h^{add} for Π is the function $h^{add}(s) := h^{add}(s, G)$ where $h^{add}(s, g)$ is the point-wise greatest function that satisfies $h^{add}(s, g) =$*

$$\begin{cases} 0 & g \subseteq s \\ \min_{a \in A, g \in add_a} c(a) + h^{add}(s, pre_a) & |g| = 1 \\ \sum_{g' \in g} h^{add}(s, \{g'\}) & |g| > 1 \end{cases}$$

Definition (h^{max}). *Let $\Pi = \langle F, A, c, I, G \rangle$ be a STRIPS planning task. The max heuristic h^{max} for Π is the function $h^{max}(s) := h^{max}(s, G)$ where $h^{max}(s, g)$ is the point-wise greatest function that satisfies $h^{max}(s, g) =$*

$$\begin{cases} 0 & g \subseteq s \\ \min_{a \in A, g \in add_a} c(a) + h^{max}(s, pre_a) & |g| = 1 \\ \max_{g' \in g} h^{max}(s, \{g'\}) & |g| > 1 \end{cases}$$

The Additive and Max Heuristics: Properties

Proposition (h^{max} is Optimistic). $h^{max} \leq h^+$, and thus $h^{max} \leq h^*$.

Proposition (h^{add} is Pessimistic). *For all STRIPS planning tasks Π, $h^{add} \geq h^+$. There exist Π and s so that $h^{add}(s) > h^*(s)$.*

→ Both h^{max} and h^{add} approximate h^+ by assuming that singleton sub-goal facts are achieved independently. h^{max} estimates optimistically by the most costly singleton sub-goal, h^{add} estimates pessimistically by summing over all singleton sub-goals.

The Additive and Max Heuristics: Properties, ctd.

Proposition (h^{max} and h^{add} Agree with h^+ on ∞). *For all STRIPS planning tasks Π and states s in Π, $h^+(s) = \infty$ if and only if $h^{max}(s) = \infty$ if and only if $h^{add}(s) = \infty$.*

→ States for which no relaxed plan exists are easy to recognize, and that is done by both h^{max} and h^{add}. Approximation is needed only for the cost of an optimal relaxed pan, if it exists.

Bellman-Ford for h^{max} and h^{add}

Bellman-Ford variant computing h^{add} for state s

new table $T_0^{add}(g)$, for $g \in F$
For all $g \in F$: $T_0^{add}(g) := \begin{cases} 0 & g \in s \\ \infty & \text{otherwise} \end{cases}$ ← Init
fn $c_i(g) := \begin{cases} T_i^{add}(g) & |g| = 1 \\ \sum_{g' \in g} T_i^{add}(g') & |g| > 1 \end{cases}$ ← relaxation
fn $f_i(g) := \min[c_i(g), \min_{a \in A, g \in \text{Cond}_a} c(a) + c_i(pre_a)]$ ← update
do forever:
  new table $T_{i+1}^{add}(g)$, for $g \in F$
  For all $g \in F$: $T_{i+1}^{add}(g) := f_i(g)$
  if $T_{i+1}^{add} = T_i^{add}$ **then** stop **endif**
  $i := i + 1$
enddo

→ Basically the same algorithm works for h^{max}, just change $\sum$ for max

Proposition. *Let $\Pi = \langle F, A, c, I, G \rangle$ be a STRIPS planning task. Then the series $\{T_i^{add}(g)\}_{i=0,\ldots}$ converges to $h^{add}(s, g)$, for all g.* (Proof omitted.)

Bellman-Ford for h^{max} in "TSP" in Australia

- F: $at(x)$ for $x \in \{Sy, Ad, Br, Pe, Ad\}$; $v(x)$ for $x \in \{Sy, Ad, Br, Pe, Ad\}$.
- A: $drive(x, y)$ where x, y have a road.

$$c(drive(x,y)) = \begin{cases} 1 & \{x,y\} = \{Sy, Br\} \\ 1.5 & \{x,y\} = \{Sy, Ad\} \\ 3.5 & \{x,y\} = \{Ad, Pe\} \\ 4 & \{x,y\} = \{Ad, Do\} \end{cases}$$

- I: $at(Sy)$, $v(Sy)$; G: $at(Sy)$, $v(x)$ for all x.

Content of Tables T_i:

i	$at(Sy)$	$at(Ad)$	$at(Br)$	$at(Pe)$	$at(Do)$	$v(Sy)$	$v(Ad)$	$v(Br)$	$v(Pe)$	$v(Do)$
0	0	∞	∞	∞	∞	0	∞	∞	∞	∞
1	0	1.5	1	5	5.5	0	1.5	1	5	5.5
2	0	1.5	1	5	5.5	0	1.5	1	5	5.5

→ $h^{max}(I) =$

Bellman-Ford for h^{add} in "TSP" in Australia

- F: $at(x)$ for $x \in \{Sy, Ad, Br, Pe, Ad\}$; $v(x)$ for $x \in \{Sy, Ad, Br, Pe, Ad\}$.
- A: $drive(x, y)$ where x, y have a road.

$$c(drive(x,y)) = \begin{cases} 1 & \{x,y\} = \{Sy, Br\} \\ 1.5 & \{x,y\} = \{Sy, Ad\} \\ 3.5 & \{x,y\} = \{Ad, Pe\} \\ 4 & \{x,y\} = \{Ad, Do\} \end{cases}$$

- I: $at(Sy)$, $v(Sy)$; G: $at(Sy)$, $v(x)$ for all x.

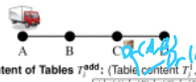
Content of Tables T_i^{add}:

i	$at(Sy)$	$at(Ad)$	$at(Br)$	$at(Pe)$	$at(Do)$	$v(Sy)$	$v(Ad)$	$v(Br)$	$v(Pe)$	$v(Do)$

→ $h^{add}(I) =$

→ $h^{add}(I) > h^+(I)$ because it counts the cost of $drive(Sy, Ad)$ 3 times:

Bellman-Ford for h^{add} in "Logistics"



- Initial state I: $t(A), p(C)$.
- Goal G: $t(A), p(D)$.
- Actions A: $dr(X, Y)$, $ld(X, a)$, $ul(Y)$.

Content of Tables T_i^{add}: (Table content T_i^1, where different, given in red)

i	$t(A)$	$t(B)$	$t(C)$	$t(D)$	$p(T)$	$p(A)$	$p(B)$	$p(C)$	$p(D)$
0	0							0	
1	0	1	2	3	3	4	5	6	7

→ $h^{\text{add}}(I) =$

→ $h^{\text{add}}(I) > h^+(I)$ because?

→ So, what if $G = \{t(D), p(D)\}$? $h^{\text{add}}(I) =$

The Additive and Max Heuristics: So What?

Summary of typical issues in practice with h^{add} and $h^{\max}$:

- Both h^{add} and $h^{\max}$ can be computed reasonably quickly.
- $h^{\max}$ is admissible, but is typically far too optimistic.
- h^{add} is not admissible, but is typically a lot more informed than $h^{\max}$.
- h^{add} is sometimes better informed than h^+, but for the "wrong reasons": rather than accounting for deletes, it overcounts by ignoring positive interactions, i.e., sub-plans shared between sub-goals.
- Such overcounting can result in dramatic over-estimates of h^+!!

→ On slide 28 with goal $t(D)$, if we have 100 packages at C that need to go to D, what is $h^{\text{add}}(I)$?

→ Relaxed plans (up next) are a means to reduce this kind of over-counting.

Relaxed Plans, Basic Idea

→ First compute a best-supporter function bs, which for every fact $p \in F$ returns an action that is deemed to be the cheapest achiever of p (within the relaxation). Then extract a relaxed plan from that function, by applying it to singleton sub-goals and collecting all the actions.

→ The best-supporter function can be based directly on $h^{\max}$ or h^{add}, simply selecting an action a achieving p that minimizes the sum of $c(a)$ and the cost estimate for pre_a.

And now for the details:

Popular Best-Supporter Functions

Definition (Best-Supporters from $h^{\max}$ and h^{add}). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state.*

The $h^{\max}$ supporter function $bs_s^{\max} : \{p \in F \mid 0 < h^{\max}(s, \{p\}) < \infty\} \mapsto A$ *is defined by* $bs_s^{\max}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\max}(s, pre_a)$.

The h^{add} supporter function $bs_s^{\text{add}} : \{p \in F \mid 0 < h^{\text{add}}(s, \{p\}) < \infty\} \mapsto A$ *is defined by* $bs_s^{\text{add}}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\text{add}}(s, pre_a)$.

Example h^{add} in "Logistics":

Heuristic Values:

	$t(A)$	$t(B)$	$t(C)$	$t(D)$	$p(T)$	$p(A)$	$p(B)$	$p(C)$	$p(D)$
h^{add}	0	1	2	3	3	4	5	0	7

Yields best-supporter function:

	$t(A)$	$t(B)$	$t(C)$	$t(D)$	$p(T)$	$p(A)$	$p(B)$	$p(C)$	$p(D)$
bs^{add}	–	$dr(A, B)$	$dr(B, C)$	$dr(C, D)$	$ld(C)$	$ul(A)$	$ul(B)$	–	$ul(D)$

Relaxed Plan Extraction

Relaxed Plan Extraction for state s and best-supporter function bs

$Open := G \setminus s$; $Closed := \emptyset$; $RPlan := \emptyset$
while $Open \neq \emptyset$ **do**:
  select $g \in Open$
  $Open := Open \setminus \{g\}$; $Closed := Closed \cup \{g\}$;
  $RPlan := RPlan \cup \{bs(g)\}$; $Open := Open \cup (pre_{bs(g)} \setminus (s \cup Closed))$
endwhile
return $RPlan$

→ Starting with the top-level goals, iteratively close open singleton sub-goals by selecting the best supporter.

This is fast! Number of iterations bounded by $|P|$, each near-constant time.

But is it correct?

→ What if $g \notin add_{bs(g)}$?

→ What if $bs(g)$ is undefined?

→ What if the support for g eventually requires g itself as a precondition?

Relaxed Plan Extraction from h^{add} in "Logistics"

- Initial state I: $t(A), p(C)$.
- Goal G: $t(A), p(D)$.
- Actions A: $dr(X,Y), ld(X), ul(X)$.

	$t(A)$	$t(B)$	$t(C)$	$t(D)$	$p(T)$	$p(A)$	$p(B)$	$p(C)$	$p(D)$
$bs^{s_0,G}$	–	$dr(A,B)$	$dr(B,C)$	$dr(C,D)$	$ld(C)$	$ul(A)$	$ul(B)$	–	$ul(D)$

Extracting a relaxed plan:

1. $bs_s^{\text{add}}(p(D)) =$
2. $bs_s^{\text{add}}(t(D)) =$
3. $bs_s^{\text{add}}(t(C)) =$
4. $bs_s^{\text{add}}(t(B)) =$
5. $bs_s^{\text{add}}(p(T)) =$
6. Anything more?

→ $h^{FF}(I) =$

→ What if $G = \{t(D), p(D)\}$? $h^{FF}(I) =$

Best-Supporter Functions

→ For relaxed plan extraction to make sense, it requires a *closed well-founded* best-supporter function:

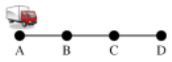
Definition (Best-Supporter Function). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state. A best-supporter function for s is a partial function $bs : (F \setminus s) \mapsto A$ such that $p \in add_a$ whenever $a = bs(p)$.*

The support graph of bs is the directed graph with vertices $F \cup A$ and arcs $\{(p, a) \mid p \in pre_a\} \cup \{(a, p) \mid a = bs(p)\}$. We say that bs is closed if $bs(p)$ is defined for every $p \in (F \setminus s)$ that has a path to a goal $g \in G$ in the support graph. We say that bs is well-founded if the support graph is acyclic.

- "$p \in add_a$ whenever $a = bs(p)$": Prerequisite (A).
- bs is closed: Prerequisite (B).
- bs is well-founded: Prerequisite (C).

→ Intuition for (C): Relaxed plan extraction starts at the goals, and chains backwards in the support graph. If there are cycles, then this backchaining may not reach the currently true state s, and thus not yield a relaxed plan.

Support Graphs and Prerequisite (C) in "Logistics"



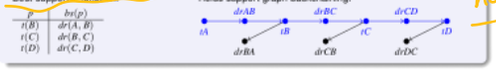
- Initial state: tA.
- Goal: tD.
- Actions: $drXY$.

How to do it (well-founded)

Best-supporter function:

p	$bs(p)$
$t(B)$	$dr(A,B)$
$t(C)$	$dr(B,C)$
$t(D)$	$dr(C,D)$

Yields support graph backchaining:

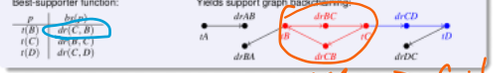


How to NOT do it (not well-founded)

Best-supporter function:

p	$bs(p)$
$t(B)$	$dr(C,B)$
$t(C)$	$dr(B,C)$
$t(D)$	$dr(C,D)$

Yields support graph backchaining:



How to obtain closed well-founded bs?

Definition (Best-Supporters from $h^{\max}$ and h^{add}). *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, and let s be a state.*

The $h^{\max}$ supporter function $bs_s^{\max} : \{p \in F \mid 0 < h^{\max}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\max}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\max}(s, pre_a)$.

The h^{add} supporter function $bs_s^{\text{add}} : \{p \in F \mid 0 < h^{\text{add}}(s, \{p\}) < \infty\} \mapsto A$ is defined by $bs_s^{\text{add}}(p) := \arg\min_{a \in A, p \in add_a} c(a) + h^{\text{add}}(s, pre_a)$.

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task such that, for all $a \in A$, $c(a) > 0$. Let s be a state where $h^+(s) < \infty$. Then both $bs_s^{\max}$ and bs_s^{add} are closed well-founded supporter functions for s.*

Proof. Since $h^+(s) < \infty$ implies $h^{\max}(s) < \infty$, it is easy to see that $bs_s^{\max}$ is closed (details omitted). If $a = bs_s^{\max}(p)$, then a is the action yielding $0 < h^{\max}(s, \{p\}) < \infty$ in the $h^{\max}$ equation. Since $c(a) > 0$, we have $h^{\max}(s, pre_a) < h^{\max}(s, \{p\})$ and thus, for all $q \in pre_a$, $h^{\max}(s, \{q\}) < h^{\max}(s, \{p\})$. Transitively, if the support graph contains a path from fact vertex r to fact vertex t, then $h^{\max}(s, \{r\}) < h^{\max}(s, \{t\})$. Thus there can't be cycles in the support graph and $bs_s^{\max}$ is well-founded. Similar for bs_s^{add}.

The Relaxed Plan Extraction: Correctness

Proposition. *Let $\Pi = (F, A, c, I, G)$ be a STRIPS planning task, let s be a state, and let bs be a closed well-founded best-supporter function for s. Then the action set $RPlan$ returned by relaxed plan extraction can be sequenced into a relaxed plan $\vec{a}^+$ for s.*

Proof. Order a before a' whenever the support graph contains a path from a to a'. Since the support graph is acyclic, such a sequencing $\vec{a} := \langle a_1, \ldots, a_n \rangle$ exists. We have $p \in s$ for all $p \in pre_{a_1}$, because otherwise $RPlan$ would contain the action $bs(p)$, necessarily ordered before a_1. We have $p \in s \cup add_{a_1}$ for all $p \in pre_{a_2}$, because otherwise $RPlan$ would contain the action $bs(p)$, necessarily ordered before a_2. Iterating the argument shows that $\vec{a}^+$ is a relaxed plan for s.

The Relaxed Plan Heuristic

Definition (Relaxed Plan Heuristic). *A heuristic function is called a relaxed plan heuristic, denoted h^{FF}, if, given a state s, it returns ∞ if no relaxed plan exists, and otherwise returns $\sum_{a \in RPlan} c(a)$ where $RPlan$ is the action set returned by relaxed plan extraction on a closed well-founded best-supporter function for s.*

→ Recall: If a relaxed plan exists, then there also exists a closed well-founded best-supporter function, see previous slide.

The Relaxed Plan Heuristic, ctd.

safe

Proposition (h^{FF} is Pessimistic and Agrees with h^+ on ∞). *For all STRIPS planning tasks Π, $h^{FF} \geq h^+$, for all states s, $h^+(s) = \infty$ if and only if $h^{FF}(s) = \infty$. There exist Π and s so that $h^{FF}(s) > h^*(s)$.*

Proof. $h^{FF} \geq h^+$ follows directly from the previous proposition. Agrees with h^+ on ∞: direct from definition. Inadmissiblity: Whenever h_S makes sub-optimal choices. → **Exercise, perhaps**

→ Relaxed plan heuristics have the same theoretical properties as h^{add}.

So what's the point?

- Can h^{FF} over-count, i.e., count sub-plans shared between sub-goals more than once?
- h^{FF} *may* be inadmissible, just like h^{add}, but for more subtle reasons.
- In practice, h^{FF} typically does not over-estimate h^* (or not by a large amount, anyway); cf. example on previous "Logistics" slide.

Helpful Actions

Definition (Helpful Actions) *Let h^{FF} be a relaxed plan heuristic, let s be a state, and let $RPlan$ be the action set returned by relaxed plan extraction on the closed well-founded best-supporter function for s which underlies h^{FF}. Then an action a applicable to s is called helpful if it is contained in $RPlan$.*

Remarks

- Initially introduced in FF [Hoffmann and Nebel (2011)], restricting Enforced Hill-Climbing to use **only** the helpful actions
- Expanding only helpful actions does not guarantee completeness.
- Other planners use helpful actions as prefered operators, expanding first nodes resulting from helpful actions.

Questionnaire

Question!

How does ignoring delete lists simplify FreeCell?

(A): You can move all cards immediately to their goal.
(B): Free cells remain free.

Question!

How does ignoring delete lists simplify Sokoban?

(A): Free positions remain free.
(B): You can walk through walls.
(C): You can push 2 stones to same position.
(D): Nothing ever becomes blocked.

Summary

- The delete relaxation simplifies STRIPS by removing all delete effects of the actions.
- The cost of optimal relaxed plans yields the heuristic function h^+, which is admissible but hard to compute.
- We can approximate h^+ optimistically by h^{max}, and pessimistically by h^{add}. h^{max} is admissible, h^{add} is not. h^{add} is typically much more informative, but can suffer from over-counting.
- Either of h^{max} or h^{add} can be used to generate a closed well-founded best-supporter function, from which we can extract a relaxed plan. The resulting relaxed plan heuristic h^{FF} does not do over-counting, but otherwise has the same theoretical properties as h^{add}; it typically does not over-estimate h^*.

Example Systems

HSP [*Bonet and Geffner, AI-01*]

1. Search algorithm: Greedy best-first search.
2. Search control: h^{add}.

FF [*Hoffmann and Nebel ,JAIR-01*]

1. Search algorithm: Enforced hill-climbing.
2. Search control: h^{FF} extracted from h^{max} supporter function; helpful actions pruning (basically expand only those actions contained in the relaxed plan).

LAMA [*Richter and Westphal, JAIR-10*]

1. Search algorithm: Multiple-queue greedy best-first search.
2. Search control: h^{FF} + a landmarks heuristic (→ similar to goal counting); for each, one search queue all actions, one search queue only helpful actions.

BFWS [*Lipovetzky and Geffner, AAAI-17*]

1. Search algorithm: best-first width search (→ next lecture).
2. Search control: novelty (→ next lecture) + variant of h^{FF} + goal counting.

Remarks

- The delete relaxation is aka ignoring delete lists.
- HSP was competitive in the 1998 International Planning Competition (IPC'98); FF outclassed the competitors in IPC'00.
- The delete relaxation is still used at large, specially since the wins of LAMA in the satisficing planning tracks of IPC'08 and IPC'11.

Remarks, ctd.

- → More generally, the relaxation principle is very generic and potentially applicable in many different contexts, as are all relaxation principles covered in this course.
- While h^{max} is not informative in practice, other lower-bounding approximations of h^+ are very important for optimal planning: admissible landmarks heuristics [*Karpas and Domshlak, IJCAI-09*]; LM-cut heuristic [*Helmert and Domshlak, ICAPS-09*].
- It has always been a challenge to take *some* delete effects into account. Recent work done to interpolate smoothly between h^+ and h^*: explicitly represented fact conjunctions [*Keyder, Hoffmann, and Haslum ICAPS-12*].

Reading

- *Planning as Heuristic Search* [*Bonet and Geffner, AI-01*].

  Available at:

  `http://www.dtic.upf.edu/~hgeffner/html/reports/hsp-aij.ps`

  Content: This is "where it all started": the first paper[1] explicitly introducing the notion of heuristic search and automatically generated heuristic functions to planning. Introduces the additive and max heuristics h^{add} and h^{max}.

[1] Well, this is the first full journal paper treating the subject; the same authors published conference papers in AAAI'97 and ECP'99, which are subsumed by the present paper.

Reading, ctd.

- *The FF Planning System: Fast Plan Generation Through Heuristic Search* [*Hoffmann:nebel:jair-01*]. JAIR Best Paper Award 2005.

  Available at:

  `http://fai.cs.uni-saarland.de/hoffmann/papers/jair01.pdf`

  Content: The main reference for delete relaxation heuristics (cited > 1000 times). Introduces the relaxed plan heuristic, extracted from the h^{max} supporter function.[2] Also introduces helpful actions pruning, and enforced hill-climbing.

[2] Done in a uniform-cost setting presented in terms of relaxed planning graphs instead of h^{max}, and not identifying the more general idea of using a well-founded best-supporter function (I used the same simpler presentation in the AI'12 core course). The notion of best-supporter functions (handling non-uniform action costs) first appears in [*Keyder and Geffner, ECAI-08*].

Reading, ctd.

- *Semi-Relaxed Plan Heuristics* [*Keyder, Hoffmann, and Haslum ICAPS-12*]. Best Paper Award at ICAPS'12.

  Available at: `http://fai.cs.uni-saarland.de/hoffmann/papers/icaps12a.pdf`

  Content: Computes relaxed plan heuristics within a compiled planning task Π^C_{ve}, in which a subset C of all fact conjunctions in the task is represented explicitly as suggested by [*Haslum, ICAPS-12*]. C can in principle always be chosen so that $h^+_{\Pi^C_{ve}}$ is perfect (equals h^* in the original planning task), so the technique allows to interpolate between h^+ and h^*. In practice, small sets C sometimes suffice to obtain dramatically more informed relaxed plan heuristics.

Chapter 6

2020年9月10日 星期四 14:44

intro06-pre-handout

COMP90054 — AI Planning for Autonomy

5. Width Based Planning

Searching for Novelty
and How to Plan with Simulators

Nir Lipovetzky

THE UNIVERSITY OF MELBOURNE

Semester 2, 2020

Agenda

Structure

Planning is **PSPACE-complete**, but current planners can **solve most of benchmarks in a few seconds**

Question:

- Can we explain why planners perform well?
- Can we characterize the line that separates '**easy**' from '**hard**' domains?

Our Answer

A new width notion and planning algorithm exponential in problem width:

- Benchmark domains have **small width** when **goals** restricted to **single atoms**
- Joint goals **easy to serialize** into a sequence of single goals

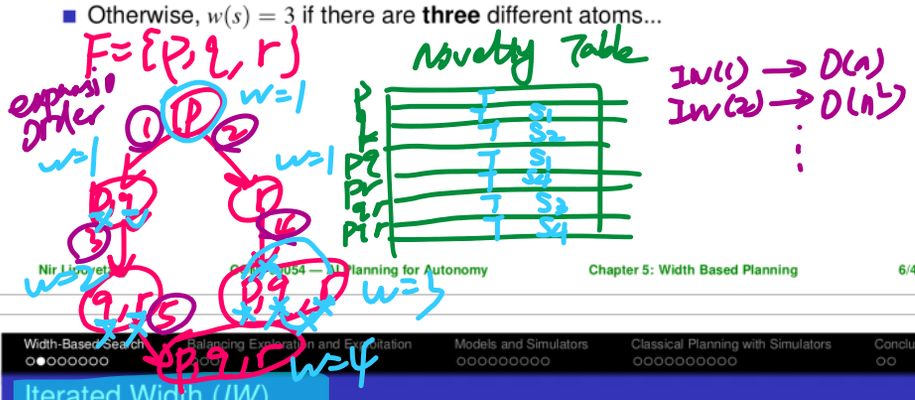
Do you want Hard Problems?

- problems with **high atomic width** (apparently no benchmark in this class)
- **multiple goal** problems that are **not easy to serialize** (e.g. Sokoban)

Definition: Novelty

Key definition: the **novelty** $w(s)$ of a state s is the size of the smallest subset of atoms in s that is true for the first time in the search.

- e.g. $w(s) = 1$ if there is **one** atom $p \in s$ such that s is the first state that makes p true.
- Otherwise, $w(s) = 2$ if there are **two** different atoms $p, q \in s$ such that s is the first state that makes $p \wedge q$ true.
- Otherwise, $w(s) = 3$ if there are **three** different atoms...



Iterated Width (*IW*)

Algorithm

- $IW(k)$ = breadth-first search that prunes newly generated states whose $novelty(s) > k$.
- IW is a sequence of calls $IW(k)$ for $i = 0, 1, 2, \ldots$ over problem P until problem solved or i exceeds number of variables in problem

Properties

$IW(k)$ expands at most $O(n^k)$ states, where n is the number of atoms.

Is IW any good in Classical Planning?

- *IW*, while simple and blind, is a pretty good algorithm over benchmarks when goals restricted to single atoms
- This is no accident, width of benchmarks domains is small for such goals

We tested domains from previous IPCs. For each instance with N goal atoms, we created N instances with a single goal

- Results quite remarkable:

# Instances	*IW*	*ID*	*BrFS*	$GBFS + h_{add}$
37921	91%	24%	23%	91%

Why *IW* does so well?

Key theory of *IW*(k) in terms of width:

Properties

For problems $\Pi \in \mathcal{P}$, where $\boldsymbol{width(\Pi) = k}$:

- $IW(k)$ solves Π in time $O(n^k)$;
- $IW(k)$ solves Π optimally for problems with uniform cost functions
- $IW(k)$ is complete for Π

Theorem

Blocks, Logistics, Gripper, and n-puzzle have a bounded width independent of problem size and initial situation, provided that goals are single atoms.

In practice, $IW(k \leq 2)$ solves 88.3% IPC problems with single goals:

# Instances	$k = 1$	$k = 2$	$k > 2$
37921	37.0%	51.3%	11.7%

IW in Classical Planning?

Primary question: *IW* solves atomic goals, how do we extend the blind procedure to multiple atomic goals?

Serialized Iterated Width (SIW)

- Simple way to **use *IW*** for solving real benchmarks P with **joint goals** is by simple form of "**hill climbing**" over goal set G with $|G| = n$, achieving atomic goals one at a time.

(Loading state space)

Serialized Iterated Width (SIW)

- ***SIW* uses *IW*** for both **decomposing** a problem into subproblems and for **solving** subproblems
- It's a **blind search** procedure, **no heuristic** of any sort, ***IW* does not even know next goal** G_i "to achieve"

Blind *SIW* better than GBFS + h_{add} !

Summary (so far)

***IW*:** *IW* sequence of novelty-based pruned breadth-first searches

- **Experiments:** excellent when goals restricted to atomic goals
- **Theory:** such problems have low width w and *IW* runs in time $O(n^w)$

***SIW*:** *IW* serialized, used to attain top goals one by one

- **Experiments:** faster, better coverage and much better plans than GBFS planner with h_{add}
- **Intuition:** goals easy to serialize and have atomic low width w

Classical Planning

State-of-the-art methods for **satisficing planning** rely on:

- heuristics derived from problem
- plugged into Greedy Best-First Search (GBFS)
- extensions (like helpful actions and landmarks)

Shortcoming of GBFS: Exploration and Exploitation

GBFS is pure greedy "exploitation"; often gets **stuck in local minima**

- Recent approaches improve performance by adding exploration

Exploration required for optimal behavior in RL and MCTS

- Such methods perform flat exploration that **ignores structure of states**

AlphaGo

eventually → global minima

We study impact of **width-based exploration methods** that take structure of states into account

state variables / proposition / atoms / facts

Best-First Width Search (BFWS)

BFWS(f)

BFWS(f) for $f = \langle w, f_1, .., f_n \rangle$ where w is a novelty-measure, is a plain best-first search where nodes are ordered in terms of novelty function w, with ties broken by functions f_i in that order.

Basic BFWS($\langle w, h \rangle$) scheme obtained with $\mathbf{h} = \mathbf{h}_{\mathbf{add}}$ or $\mathbf{h}_{\mathbf{ff}}$, and novelty-measure $\mathbf{w} = \mathbf{w}_{\mathbf{h}}$, where

- $w_h(s)$ = size of smallest new tuple of atoms generated by s for the first time in the search relative to previously generated states s' with $h(s) = h(s')$.

→ BFWS($\langle w, h \rangle$) much better than purely greedy BFS(h)

Some BFWS(f) variants yield state-of-the-art performance:

- 1st place in Agile track, Runner-Up Satisficing track IPC-2018
- more info: `https://ipc2018-classical.bitbucket.io/`

Classical Planning

The status quo:

- Model usually represented in compact form (STRIPS, PDDL)

Introduction & Motivation

For more than 40 years, research focused on **exploiting** information about **action preconditions and effects** in order to plan efficiently

- GPS, POP, GraphPlan, SAT, OBDD, heuristic-search planning, ...

We showed that same level of efficiency can be obtained with **simulators**: **without** a representation of **action preconditions and effects**

Novelty: $|F|$

Introduction & Motivation

This has been shown by:

- Developing a planner that uses action structure **only** to define
  - the set A(s) of **applicable actions** in state s, and
  - state **transition function** f(a,s)

  The **planner does not see action preconditions and effects** but just the functions A(s) and f(a,s)
- We showed that its **performance matches** the performance of state of the art **planners that make use of PDDL** representations, over the existing PDDL benchmarks

Many consequences follow from this radical departure

Modeling

Modeling

Many problems fitting **classical planning** model but **difficult to describe in PDDL** are easily modeled now: Pacman, Tetris, Pong, etc.

- **Expressive language features** easily supported: functions, conditional effects, derived predicates, state constraints, quantification, ...
- Any element of the problem can be modeled through logical symbols attached to **external procedures** (e.g. C++).
- Action effects can be given as **fully-black-box procedure** taking the state as input.

Introduction & Motivation

Declarative languages also have their **downsides**:

- **Model** $\neq$ **Language.** Many problems fit Classical Planning model, but hard to express in PDDL-like languages.
- Recent development of **simulation platforms** such as
  - Atari Learning Environment,
  - GVG-AI,
  - Universe, etc.

Need for planners that work **without complete declarative representations.**

Algorithm: Simulated BFWS

Framework **Best-first width search** (**BFWS**):

- Novelty measures w also used in best-first algorithms (BFWS)
- Best results when w-measures combined with goal directed heuristics h (Lipovetzky and Geffner, 2017)
- BFWS(h) picks node from OPEN with least w_h measure, breaking ties with h
  - $w_h(s) = k$ if s is first state to make some set of k atoms true, among those with heuristic $h = h(s)$

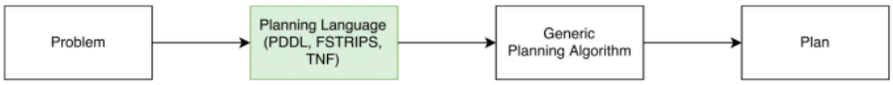
BFWS(h) much better than standard BFS(h)

Use of heuristics couples algorithm with *declarative representations*.

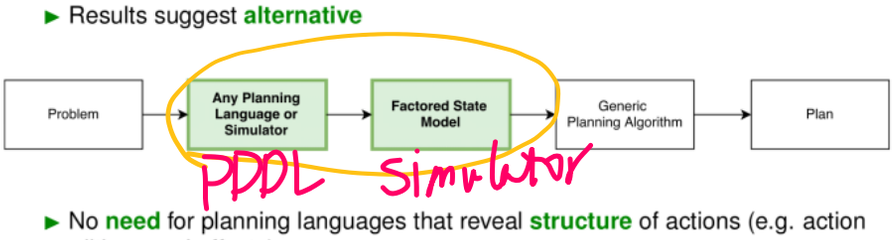
- In (Frances et al. 2017) we lift this requirement

Implications: Modeling and Control

- **Traditional** toolchain



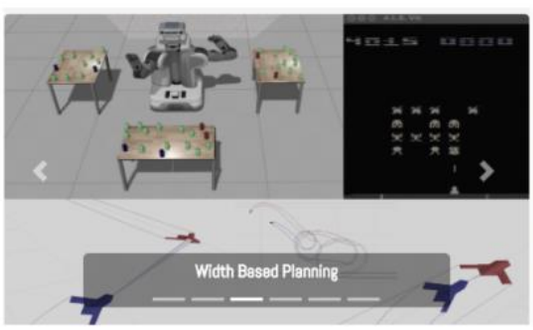
- Results suggest **alternative**



- No **need** for planning languages that reveal **structure** of actions (e.g. action preconditions and effects)
- **Not much** efficiency appears to be **lost** in **second** pathway

These algorithms open up **Exciting possibilities** for **modeling beyond PDDL**

Width-Based planning over Simulators



Challenges:

- Non-linear dynamics
- Perturbation in flight controls
- Partial observability
- Uncertainty about opponent strategy

Classical Planning with Simulators

Can classical planners work without PDDL?

Atari 2600

ATARI

Arcade Learning Environment

http://www.arcadelearningenvironment.org/
The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows researchers and hobbyists to develop *AI agents for Atari 2600 games*.

Recent AI agents:

- ⇝ Reinforcement Learning and Deep Learning trained to learn a controller
- ⇝ Search algorithm as a lookahead for action selection

Motivation

While planning is the model-based approach to control, planning research is heavily fragmented

- Many models (classical; MDPs, POMDPs; "logical" variants FOND and POND; time, resources, ..)
- Modeling languages vs. Use of Simulators
- Different communities (ICAPS-AAAI; NIPS-UAI-RL ..)

Best way to get communication across and to build on each others' work is common benchmarks and environments such as *ALE*

ALE and Classical Planning

Planning setting in ALE is deterministic and initial state fully known

Yet classical planners can't be used

- no PDDL encoding
- no goals but rewards

→ *Bellemare et al.* consider Breadth-first Search (*BrFS*) and *MCTS (UCT)*

→ Still, "classical" planning algorithms such as *IW* can be applied almost off-the-shelf!

IW in the Atari Games

novelty ← IF|

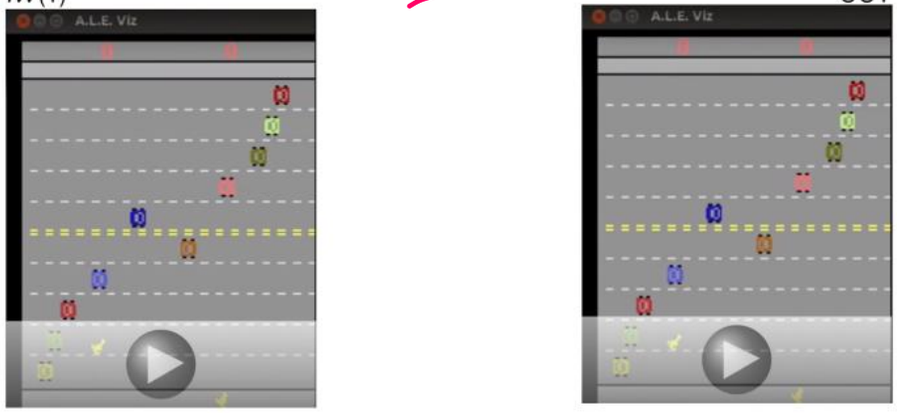
- *IW*(1) used with the 128 variables (bytes) of 256 values each
- *IW*(1) generates then up to $128 \times 256 \times 18$ (i.e, $589,824$) states
  - Children in *IW*(1) generated in random order
  - Discount factor used $\gamma = 0.995$
- Action leading to most rewarding *IW*(1)-path is executed

IW Playing Atari!

(algorithms in action!)

Freeway

IW(1) > UCT



Experimental Results

Same setting from Bellemare et al:

- Games are played for 5 minutes maximum (18,000 frames)
- 2BFS and *IW* have a maximum lookahead budget of $150,000$ simulated frames
- UCT has same budget by running 500 rollouts of depth 300
- Score is averaged among 5 runs per game

	IW(1)	2BFS	*BrFS*	UCT
# Times Best (54 games)	26	13	1	19
# Times Better than *IW*	–	16	1	19
# Times Better than 2BFS	34	–	1	25
# Times Better than UCT	31	26	1	–

Search Tree Depths

- *BrFS* search tree results in a lookahead of 0.3 seconds
- *IW*(1) and 2BFS result in lookahead of up to 6–22 seconds

IW vs DeepMind

Lookahead Agents VS Learning Agents:

⇝ Still open how to compare them best as they solve different control problems, and different inputs (RAM vs Screen)

But, taking into account gameplay score:

⇝ IW outperforms DeepMind's algorithm in 45 out of 49 games
⇝ Similar results have been reported recently over Screen Inputs [AAAI 2018]

ALE Wrap up

- *IW* makes use of the state structure (atoms) to order exploration
- *IW*(1) is a *BrFS* that keeps states that generate new atoms
- Exploitation of this structure pays off in classical planning and ALE
- First classical planners using simulators
- Youtube videos: ▸ Link http://bit.ly/1EuCb9x

The Width Conspirators: What's next?

→ Explore new **applications**: Social Sciences, Computational Sustainability, and any other fields rely on simulators

→ Width-based for other **planning computational models**: Uncertainty, Beliefs, Multi-agent

→ Devise new **algorithms** for real-time behavior

→ Bridge connection between **Control, Planning and Learning**

Help us grow the boundaries of AI planning research!

Resources

Literature:

- `https://nirlipo.github.io/Width-Based-Planning-Resources/`

LAPKT stands for the Lightweight Automated Planning ToolKiT:

- `http://lapkt.org`

IW-ALE, BFWS source code:

- `http://lapkt.org/index.php?title=Projects`

Width-based in action featured in ICAPS-19:

- `https://icaps19.icaps-conference.org/`!

Chapter 7

2020年9月10日 星期四 14:45

intro06b-PR-pre-ha...

COMP90054 — AI Planning for Autonomy

1. Plan & Goal Recognition

Contents of the Lecture

Nir Lipovetzky

THE UNIVERSITY OF MELBOURNE

Semester 2, 2020

Outline of the Lecture

The Heider-Simmel Experiment

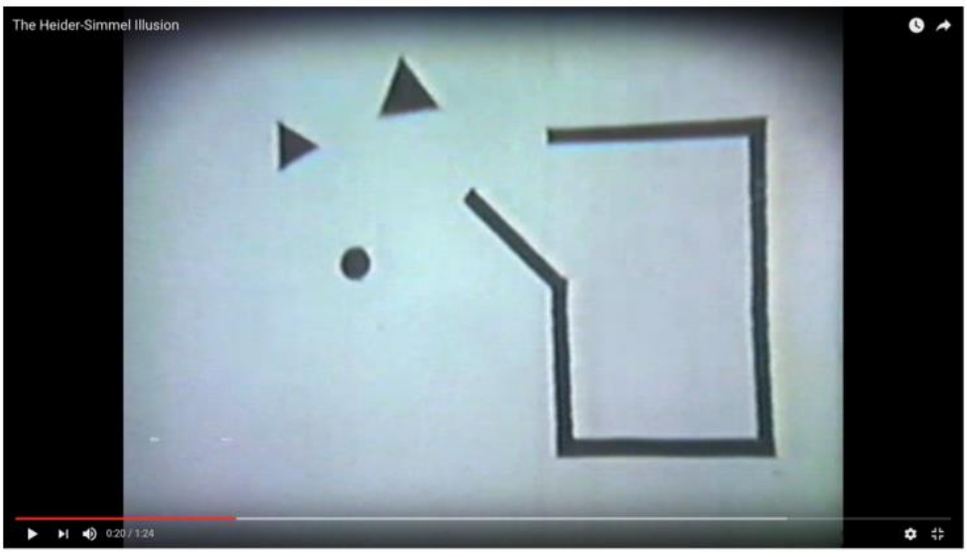


Figure: *An Experimental Study of Apparent Behavior*. F. Heider, M. Simmel. The American Journal of Psychology, Vol. 57, No. 2, April 1944

Link to video (YouTube)

Parsing the Big Triangle

Figure: The BIG triangle T.

PollEv.com/nirlipo

Question!

What kind of person is the Big Triangle?

(A): Aggressive, mean, angry.
(B): Strong, powerful.
(C): Dumb, stupid.
(D): Ugly, sly.

what about the Smaller one...

Figure: The small triangle t.

PollEv.com/nirlipo

Question!

What kind of person is the Small Triangle?

(A): Fearless, defiant, cocky.
(B): Passive–aggressive.
(C): Clever, weak.
(D): Protective, loyal, devoted.

and about the circle...

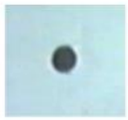


Figure: The circle c.

PollEv.com/nirlipo

Question!

What kind of person is the Circle?

(A): Frightened, fearful, helpless.
(B): Fidgety, playful, nervous.
(C): Clever, smart.
(D): Courageous.

Significance of Heider & Simmel Results

Leaving aside issues with *priming* experimental subjects...

It does seem that

1. humans *tend* to **ascribe intentions** to *anything* that changes over time,
2. this rests on deeply rooted **assumptions**.

Heider & Simmel results are the *first* quantitative characterization of:

Folk Psychology

Human capacity to **explain** and **predict** behavior and mental state of others

...we're usually very good at it, but we fail often!

A Theory of Common Sense

The Intentional Stance, **Daniel Dennett** (1988)

1. **Decide** to consider the object being observed as *rational*
2. Work out its **beliefs** and **goals** based on its *place* and *purpose* in the world.
3. Use **practical reasoning** to assess what the agent *ought to do* to pursue its **goals**.

The above provides a *systematic*, *reason–giving explanation* for actions, based on deeply embedded beliefs about the agent.

planning

Plan and Goal Recognition in Artificial Intelligence

Key Idea: use generative models of behavior to predict actions.

Plan Recognition (PR) is Planning in reverse.

- Planning – we seek *plans* π to *achieve* goals G.
- PR: find goals G *accounting for* partially *observed* plan π.

Formalising GR as a Multi–Agent Task

Two possible *roles* for each agent:

- **Actor** – *performs* actions to change the state of the world.
- **Observer** – *perceives* actions and updates its beliefs on the **Actor** intentions.

and three possible *stances* for the **Actor**:

- *Adversarial* – obfuscates deliberately its goals.
- *Cooperative* – tries to tell the **Observer** what she is up to.
- *Indifferent* – does not care about the **Observer**.

Open Challenge→Stances could be *changing over time*

Components of Goal Recognition Task

Actions describe what the **Actor** does

- Walking from X to Y, opening a door, using a credit card...

Goals describe what the **Actor** wants

- To have breakfast, Park a car, Wreck a web service...

Plans describe how goals can be achieved

- Ordered sequences of actions
- These can be ranked according to **cost** or **efficiency**

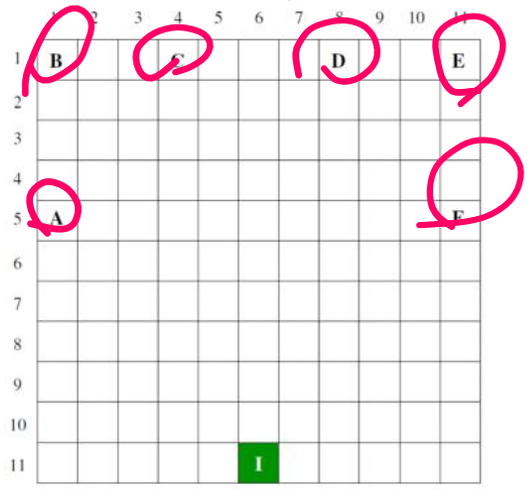
Sensor Model describes what does the **Observer** perceives

- Does it always see every action done by the **Actor**?
- Are actions observed *directly*? Or only their *effects* are?
- Does it know exactly *where* in the world the **Actor** is?

Goal Recognition can be modeled using STRIPS

Example: Agent on a Grid World

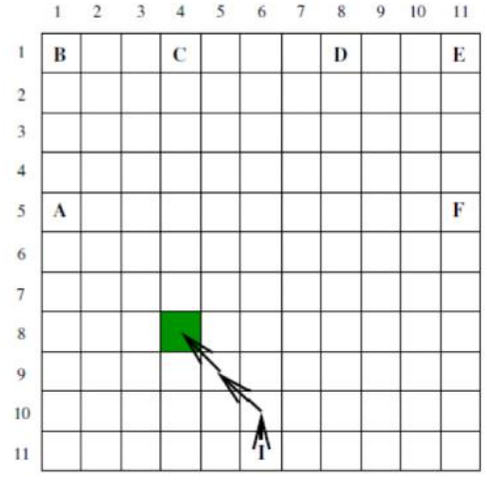
possible goals



- **starts** in "I", *may be* heading to "A", "B", ..., "F".
- **moves along compass directions** *North*, etc. with cost 1 and *North West*, etc. with cost $\sqrt{2}$.

Example

Actor now at $(4, 8)$ after going N once, and twice NW.



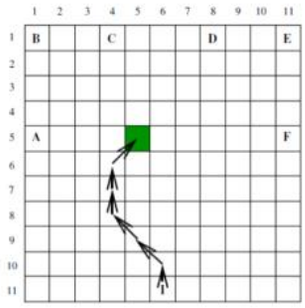
Question!

Assuming the Actor prefers CHEAPEST plans which goals are most likely?

(A): A & B. (B): C.
(C): D (D): E & F

Example

Actor now at $(5, 5)$ after going N twice and once NE.

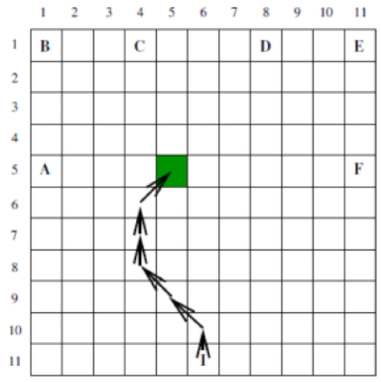


Question!

For which goal(s) observed actions are in a CHEAPEST plan?

(A): A & B.
(B): C.
(C): D, E & F
(D): None

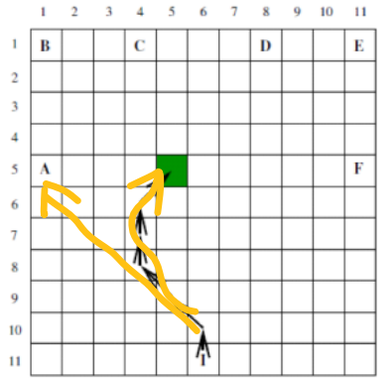
So Folk Psychology is Useless?



Remarks

- Verify obs *sufficient* for G Easy
- Determine to what degree obs *necessary* for G Hard

Folk Psychology with Counterfactual Reasoning



Counterfactual Reasoning (Pearl, 2001) to Establish Necessity

Compare **cost** of best plans that do not comply with observed actions, with best plans that do.

→ Then it follows B and C *more likely* than A or the rest.

Key Facts of the Model–Based Approach

1. Π given **implicitly**, requires to **solve** $|\mathcal{G}|$ planning tasks
2. Plans "**extracted**" with **off–the–shelf** planning algorithms.
3. **Plausibility** of goals $\mathcal{G}$ given as a **probability distribution**
   - Goals are *plausible* when motivate plans *consistent* with O,
   - **and** when O is *necessary* to achieve goals *efficiently*.

Roadmap

1. Make off-the-shelf planners compute plans constrained w.r.t. O,
2. Derive $P(G|O)$ from best plans that comply with *and* work around O.

PR as planning: Inferring the Goal Probabilities

Goal

Obtain **probability distribution** $P(G|O)$, $G \in \mathcal{G}$.

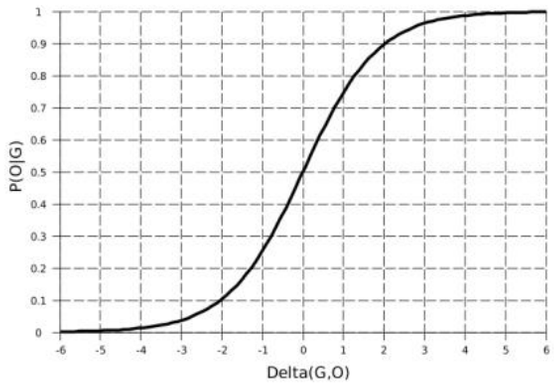
Outline of Approach

From **Bayes' Rule** $P(G|O) = \alpha\, P(O|G)\, Prob(G)$, where

- α norm. constant
- $Prob(G)$ given in problem specification
- $P(O|G)$ function of extra cost needed to not comply with O

$$P(O|G) = \text{function}(c^*(P'[G + \overline{O}])) - c^*(P'[G + O])) \quad (1)$$

Goals as Predictors for O (informally)



Properties

1. G predicts O badly when it would be more efficient to deviate from O.
2. G predicts O **perfectly** when G unfeasible if not doing O.

Demo: A Slightly More Interesting STRIPS Model

Fluents: *facts about the world*
- Locations of people
- State of appliances
- Locations of objects

Actions: *stuff people may do*
- Move across the place
- Interaction with objects & appliances

Goals: *why people do stuff*
- Cook some foodstuff
- Watch a movie
- Listen to a record
- Go to sleep
- Get ready to leave for work

Unitary action costs (to keep it simple)

GitHub Repo Pull Requests Welcome!

Anyone looking for a Masters' project? Thor 2 has been released!

Further Reading or Watching

Article *An Experimental Study of Apparent Behavior*. F. Heider, M. Simmel. The American Journal of Psychology, 57(2), 1944
A Probabilistic Plan Recognition Algorithm based on Plan Tree Grammars C. Geib, R. Goldman, Artificial Intelligence 173(11), 2009
Probabilistic Plan Recognition using off-the-shelf Classical Planners. M. Ramirez and H. Geffner. Proceedings AAAI, 2010.
Landmark-Based Heuristics for Goal Recognition. R. Pereira, N. Oren and F. Meneguzzi. Proceedings AAAI, 2017.
Heuristic Online Goal Recognition in Continuous Domains, M. Vered and G. Kaminka. Proceedings IJCAI, 2017.
Plan Recognition in Continuous Domains, G. Kaminka and M. Vered and N. Agmon, Proceedings AAAI, 2018.

Book Chapter 4, Section 4.3 *A Concise Introduction to Models and Methods for Automated Planning*. B. Bonet & H. Geffner, 2013.

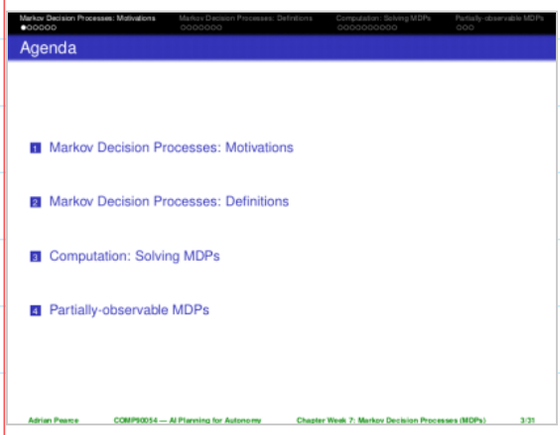
Video Lecture *Engineering & Reverse-engineering Human Common Sense*, J. Tenenbaum, Allen Institute for AI, 2015.

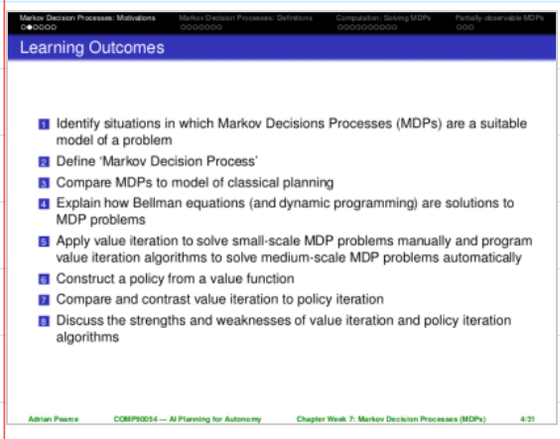
Video Lecture *Steps towards Collaborative Dialogue*, P. Cohen, Monash University, 2018.

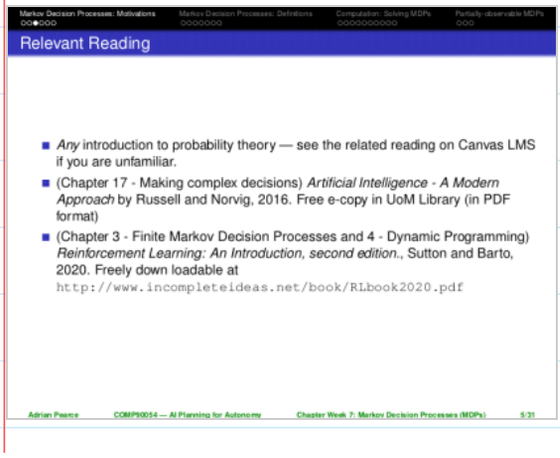
Chapter 8

2020年9月17日 星期四 14:24

intro07-pr
e-handout







Removing Assumptions

Classical Planning

So far, we have looked at classical planning. Classical planning tools can produce solutions quickly in large search spaces; but assume:

- Deterministic actions
- Environments change only as the result of an action
- Perfect knowledge (omniscience)
- Single actor (omnipotence)

Throughout the remainder of this subject, we are going to look at how to relax a few of these assumptions, starting with the first: deterministic actions.

A non-deterministic action is an action with one or more non-deterministic effects.

Markov Decision Processes

Markov Decision Processes (MDPs) remove the assumption of deterministic actions and instead assume that each action can have multiple outcomes (effects), with each outcome associated with a probability.

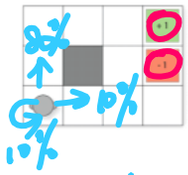
For example:

- Flipping a coin has two outcomes: heads ($\frac{1}{2}$) and tails ($\frac{1}{2}$)
- Rolling two dices together has twelve outcomes: 2 ($\frac{1}{36}$), 3 ($\frac{1}{18}$), 4 ($\frac{1}{36}$), ..., 12 ($\frac{1}{36}$)
- When trying to pick up an object with a robot arm, there could be two outcomes: successful ($\frac{4}{5}$) and unsuccessful ($\frac{1}{5}$)

MDPs have been successfully applied to planning in many domains: robot navigation, planning which areas of a mine to dig for minerals, treatment for patients, maintenance scheduling on vehicles, and many others.

Canonical example: Grid World

Our agent is in the bottom left cell of a grid. The grey square is a wall. The two labelled cells give a *reward*: 1 for reaching the top-right cell, but a negative reward of -1 for the cell immediately below.



But! Things can go wrong — sometimes the effects of the actions are not what we want:

- If the agent tries to move north, 80% of the time, this works as planned (provided the wall is not in the way)
- 10% of the time, trying to move north takes the agent west (provided the wall is not in the way);
- 10% of the time, trying to move north takes the agent east (provided the wall is not in the way)
- If the wall is in the way of the cell that would have been taken, the agent stays put.

non-deterministic outcome

Discounted Reward Markov Decision Processes

MDPs are **fully observable, probabilistic** state models. The most common formulation of MDPs is a *Discounted-Reward* Markov Decision Process:

- a state space S
- initial state $s_0 \in S$
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- **transition probabilities** $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- **rewards** $r(s, a, s')$ positive or negative of transitioning from state s to state s' using action a
- a **discount factor** $0 \leq \gamma < 1$

What is different from classical planning? Four things:

- The transition function is no longer deterministic. Each action has a probability of $P_a(s'|s)$ of ending in state s' if a is executed in the state s.
- There are no goals. Each action receives a reward when applied. The value of the reward is dependent on the state in which it is applied.
- There are no action costs. Instead, these are modelled as negative rewards.
- We have a *discount factor*.

MDP
- Set of state S
- Initial state I
- Probabilistic state transitions
  $P_a(s'|s) \in [0,1]$
- Reward function $r(s,a,s')$
  in real
- discount factor γ
  $[0,1)$

Classical planning
- Set of state S
- Initial state I
- Transition function A
- Goals G
- costs

Discounted rewards

The discount factor determines how much a future reward should be discounted compared to a current reward.

For example, would you prefer $100 today or $100 in a year's time? We (humans) often *discount* the future and place a higher value on nearer-term rewards.

Assume our agent receives rewards $r_1, r_2, r_3, r_4, \ldots$ in that order. If γ is the discount factor, then the discounted reward is:

$$V = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots$$
$$= r_1 + \gamma(r_2 + \gamma(r_3 + \gamma(r_4 + \ldots)))$$

If V_t is the value received at time-step t, then $V_t = r_t + \gamma V_{t+1}$

MDP: Example action

Probabilistic PDDL is one way to represent an MDP. It extends PDDL with a few additional constructs. Of most relevance is that outcomes can be associated with probabilities. The following describes the "Bomb and Toilet" problem, in which one of two packages contains a bomb. The bomb can be diffused by dunking it into a toilet, but there is a 0.05 probability of the bomb clogging the toilet.

```
(define (domain bomb-and-toilet)
  (:requirements :conditional-effects :probabilistic-effects)
  (:predicates (bomb-in-package ?pkg) (toilet-clogged)
               (bomb-defused))
  (:action dunk-package
    :parameters (?pkg)
    :effect (and (when (bomb-in-package ?pkg)
                 (bomb-defused))
                 (probabilistic 0.05 (toilet-clogged))))
```


Solutions for MDP problems: policies

The planning problem for discounted-reward MDPs is different to that of classical planning because the actions are non-deterministic. Instead of a sequence of actions, an MDP produces a *policy*.

A policy, π is a function that tells an agent which is the best action to choose in each state. A policy can be *deterministic* or *stochastic*.

A deterministic policy $\pi : S \to A$ is a *mapping* from states to actions. It specifies which action to choose in every possible state. Thus, if we are in state s, our agent should choose the action defined by $\pi(s)$.

A graphical representation of the policy for Grid World is:

→	→	→	+1
↑		↑	−1
↑	←	↑	←

So, in the initial state (bottom left cell), following this policy the agent should go up.

Solutions for MDP problems: policies (continued)

Of course, agents do not work with graphical policies. The output from an MDP planner would look more like this:

```
at(0,0) => move_up
at(0,1) => move_up
at(0,2) => move_right
at(1,0) => move_left
at(1,2) => move_right
at(2,0) => move_up
at(2,2) => move_right
at(3,0) => move_left
```

An agent can then parse this in and use it by determining what state it is in, looking up the action for that state, and executing the action. Then repeat.

Stochastic policies

A stochastic policy $\pi : S \times A \to \mathbb{R}$ specifies the *probability distribution* from which an agent should select an action. Intuitively, $\pi(s, a)$ specifies the probability that action a should be executed in state s.

To execute a stochastic policy, we could just take the action with the maximum $\pi(s, a)$. However, in many domains, it is better to select an action based on the probability distribution; that is, choose the action probablistically such that actions with higher probability are chosen proportionally to their relative probabilities.

In this subject, we will focus only on deterministic policies, but stochastic policies have their place too.

Expected Value of Policy for Discounted Reward MDPs

For discounted-reward MDPs, optimal solutions maximise the *expected discounted accumulated reward* from the initial state s_0. But what is the expected discounted accumulated reward?

- In Discounted Reward MDPs, the **expected discounted reward from** s is

$$V^{\pi}(s) = E_{\pi}[\sum_{i} \gamma^{i} r(a_i, s_i) \mid s_0 = s, a_i = \pi(s_i)]$$

Thus, $V^{\pi}(s)$ defines the expected value of following the policy π from state s.

So for our Grid World example, assuming only the -1 and +1 states have rewards, the expected value is:

$\gamma^5 \times 1 \times (0.8^5)$ (optimal movement)
$\gamma^7 \times 1 \times (0.8^7)$ (first move only fails)
$\ldots$ (etc.)

Bellman equations (Discounted-Reward MDPs)

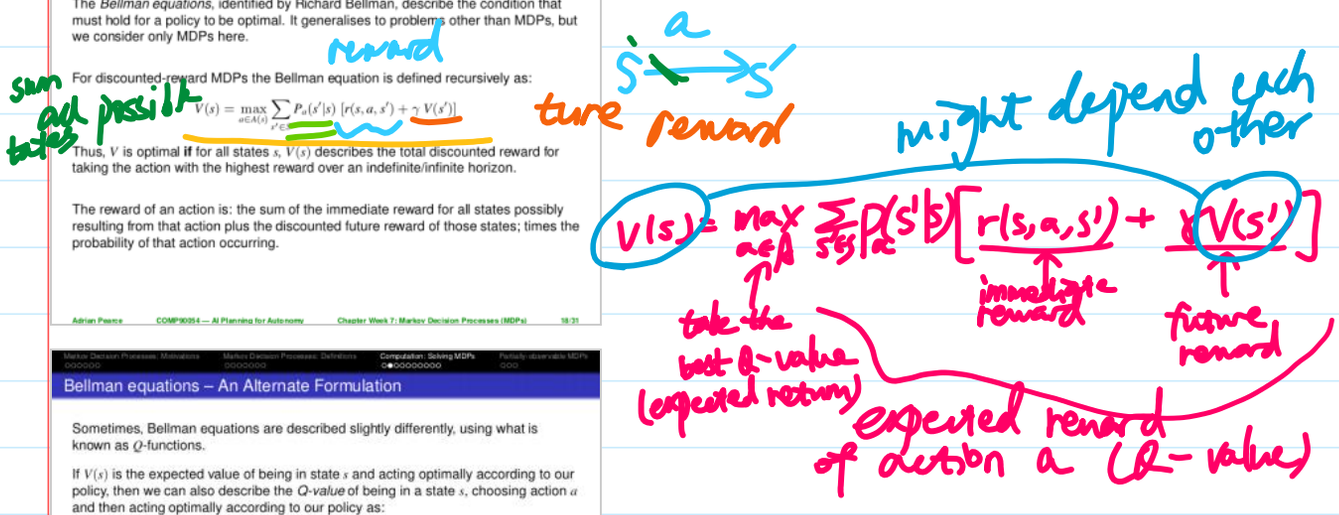
The *Bellman equations*, identified by Richard Bellman, describe the condition that must hold for a policy to be optimal. It generalises to problems other than MDPs, but we consider only MDPs here.

For discounted-reward MDPs the Bellman equation is defined recursively as:

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \, [r(s, a, s') + \gamma \, V(s')]$$

Thus, V is optimal **if** for all states s, $V(s)$ describes the total discounted reward for taking the action with the highest reward over an indefinite/infinite horizon.

The reward of an action is: the sum of the immediate reward for all states possibly resulting from that action plus the discounted future reward of those states; times the probability of that action occurring.



Bellman equations – An Alternate Formulation

Sometimes, Bellman equations are described slightly differently, using what is known as Q-functions.

If $V(s)$ is the expected value of being in state s and acting optimally according to our policy, then we can also describe the *Q-value* of being in a state s, choosing action a and then acting optimally according to our policy as:

For discounted-reward MDPs the Bellman equation is defined recursively as:

$$Q(s, a) = \sum_{s' \in S} P_a(s'|s) \, [r(s, a, s') + \gamma \, V(s')]$$

This is just the expression inside the $\max$ expression in the Bellman equation. Using this, sometimes you may see the Bellman equation then defined as:

$$V(s) = \max_{a \in A(s)} Q(s, a)$$

The two definitions are equivalent, but you may seem them defined in both ways. However, when we move onto reinforcement learning later, we will use Q functions more explicitly.

Using the Bellman Equations: Value Iteration

Value Iteration finds the optimal value function V^* solving the Bellman equations iteratively, using the following algorithm:

- Set V_0 to arbitrary value function; e.g., $V_0(s) = 0$ for all s.
- Set V_{i+1} to result of Bellman's **right hand side** using V_i in place of V_i:

$$V_{i+1}(s) := \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \left[r(s,a,s') + \gamma V_i(s') \right]$$

This converges exponentially fast to the optimal policy as iterations continue.

Theorem

$V_i \mapsto V^*$ as $i \mapsto \infty$. That is, given an infinite amount of iterations, it will be optimal.

The complexity of each iteration is $O(|S|^2|A|)$. On each iteration, we iterate in an outer loop over all states in S, and in each outer loop iteration, we need to iterate over all states ($\sum_{s' \in S}$), meaning $|S|^2$ iterations. But also within each outer loop iteration, we need to calculate the value for every action to find the maximum.

dynamic programming

use previous value

Value Iteration in Practice

Value Iteration converges to the optimal value function V^* asymptotically, but in practice, the algorithm is stopped when the **residual** $R = \max_s |V_{i+1}(s) - V_i(s)|$ reaches some pre-determined threshold ϵ – that is, when the largest change in the values between iterations is "small enough".

The resulting greedy policy π_V has it's **loss** bounded by $2\gamma R/1 - \gamma$.

It is clear to see that the value iteration can be easily parallelised by updating the value of many states at once: the values of states at step $t+1$ are dependent only on the value of other states at step t.

A policy can now be easily defined: in a state s, given V, choose the action with the highest expected reward.

Value iteration example: Grid World.

Assuming $\gamma = 0.9$.

After 1 iteration

0.00	0.00	0.00	+1
0.00		0.00	−1
0.00	0.00	0.00	0.00

After 2 iterations

0.00	0.00	0.72	+1
0.00		0.00	−1
0.00	0.00	0.00	0.00

After 3 iterations

0.00	0.52	0.78	+1
0.00		0.43	−1
0.00	0.00	0.00	0.00

After 4 iterations

0.37	0.66	0.83	+1
0.00		0.51	−1
0.00	0.00	0.31	0.00

After 5 iterations

0.51	0.72	0.84	+1
0.27		0.55	−1
0.00	0.22	0.37	0.13

After 100 iterations

0.64	0.74	0.85	+1
0.57		0.57	−1
0.49	0.43	0.48	0.28

After 1000 iterations

0.64	0.74	0.85	+1
0.57		0.57	−1
0.49	0.43	0.48	0.28

Policy after 1000 iterations

→	→	→	+1
↑		↑	−1
↑	←	↑	←

converge 以 stop

Deciding How to Act

Given a policy that is (close to) optimal, how should we then select the action to play in a given state? It is reasonably straightforward: select the action that maximises our expected utility. So, given a value function V, we can select the action with the highest expected reward using:

$$\operatorname{argmax}_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \left[r(s,a,s') + \gamma V(s') \right]$$

This is known as *policy extraction*, because it extracts a policy for a value function (or Q-function). This can be calculated 'on the fly' at runtime.

Alternatively, given a Q-function instead of a value function, we can use:

$$\operatorname{argmax}_{a \in A(s)} Q(s,a)$$

This is simple to decide than using the value functions because we do not need to sum over the set of possible output states.

Policy Iteration

The other common way that MDPs are solved is using *policy iteration* – an approach that is similar to value iteration. While value iteration iterates over value functions, policy iteration iterates over policies themselves, creating a strictly improved policy in each iteration (except if the iterated policy is already optimal).

Policy iteration first starts with some (non-optimal) policy, such as a random policy, and then calculates the value of each state of the MDP given that policy — this step is called the *policy evaluation*. It then updates the policy itself for every state by calculating the expected reward of each action applicable from that state.

The basic idea here is that policy evaluation is easier to compute than value iteration because the set of actions to consider is fixed by the policy that we have so far.

Policy evaluation

The *expected reward* of policy π from s to goal, $V^\pi(s)$, is weighted avg of reward of the possible state sequences defined by that policy times their probability given π.

However, the expected reward $V^\pi(s)$ can also be characterised as a solution to the equation

$$V^\pi(s) = \sum_{s' \in S} P_a(s'|s)\,[r(s,a,s') + \gamma\, V^\pi(s')]$$

where $a = \pi(s)$, and $V^\pi(s) = 0$ for goal states

This corresponds to a set of *linear equations* which can also be solved analytically using efficient numerical methods (which are also iterative) with software tools such as MATLAB, CPLEX or Gurobi—however we do not cover these in this subject.

The *optimal expected reward* $V^*(s)$ is $\max_\pi V^\pi(s)$ and the *optimal policy* is the $\arg\max$

Policy Iteration

Let $Q^\pi(a,s)$ be the expected reward from s when doing a first and then following the policy π:

$$Q^\pi(a,s) = \sum_{s' \in S} P_a(s'|s)\,[r(s,a,s') + \gamma\, V^\pi(s')]$$

When $Q^\pi(a,s) < Q^\pi(\pi(s),s)$, π *strictly improved* by changing $\pi(s)$ to a

Policy Iteration computes π^* by a sequence of policy evaluations and improvements:

1. Starting with arbitrary policy π
2. Compute $V^\pi(s)$ for all s (policy evaluation)
3. Improve π by setting $\pi(s) := \text{argmax}_{a \in A(s)}\, Q^\pi(a,s)$ (improvement)
4. If π changed in 3, go back to 2, else *finish*

This algorithm finishes with an optimal π^* after a finite number of iterations, because the number of policies is finite, bounded by $O(|A|^{|S|})$, unlike value iteration, which can theoretically require infinite iterations.

However, each iteration costs $O(|S|^2|A| + |S|^3)$. Empirical evidence suggests that the most efficient (value iteration versus policy iteration) is dependent on the particular MDP model being solved.

{ policy evaluation
policy update

The Curse of Dimensionality

Thus, solving MDPs using these algorithms is polynomial in the size of the state space, but *exponential in the number of variables* if we use a PDDL-like language to represent our problem.

- That is, if there are N number of variables, each a Boolean, there are 2^N number of states. Value iteration and policy iteration require us to keep a vector of size $|2^N|$.

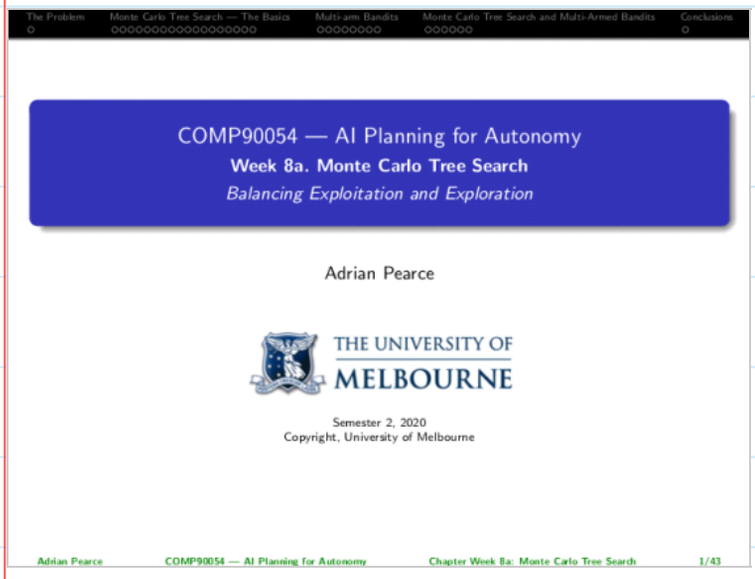
Question: Can we do better?

Answer: Yes! Using function approximation, which we will see in a couple of weeks

Chapter 9

2020年9月24日 星期四 14:39

intro8a-pr
e-handout



Learning Outcomes

1. Explaining the difference between offline and online planning for MDPs.
2. Apply MCTS solve small-scale MDP problems manually and program MCTS algorithms to solve medium-scale MDP problems automatically
3. Construct a policy from Q-functions resulting from MCTS algorithms
4. Select and apply multi-armed bandit algorithms
5. Integrate multi-armed bandit algorithms (including UCB) to MCTS algorithms
6. Compare and contrast MCTS to value/policy iteration
7. Discuss the strengths and weaknesses of the MCTS family of algorithms.

Relevant Reading

- Chapter 2 Multi-armed Bandits; Chapter 5 Monte Carlo Methods; and Section 8.11 Monte Carlo Tree Search of *Reinforcement Learning: An Introduction, second edition*, Sutton and Barto 2020. Freely downloadable at `http://www.incompleteideas.net/book/RLbook2020.pdf`
  - → Text book covering key concepts
- *A Survey of Monte Carlo Tree Search Methods* by Browne et al. *IEEE Transactions on Computational Intelligence and AI in Games*, 2012 Available at `https://ieeexplore.ieee.org/document/6145622` (Unimelb library access required—you can use the *Library Access* plugin in Chrome to automatically gain access)
  - → Good "entry level" resource, with lots of pointers to seminal papers
- Regret Analysis of Stochastic and Non-stochastic Multi-armed Bandit Problems by S. Bubeck and N. Cesa–Bianchi, 2012 `http://arxiv.org/pdf/1204.5721v2.pdf`
  - → All you want to know about regret analysis and multi-armed bandits

Offline Planning & Online Planning over MDPs

We saw value iteration and policy iteration in the previous lecture. These are **offline planning** methods, in that we solve the problem offline for all possible states, and then use the solution (a policy) online. In the offline planning approach:

- We can define policies, π, that work from any state in a convenient manner,
- Yet the state space S is usually *far* too big to determine $V(s)$ or π exactly.
- There *are* methods to approximate the MDP by reducing the dimensionality of S, but we will not discuss these until later.

In **online planning**, planning is undertaken immediately before executing an action. Once an action (or perhaps a sequence of actions) is executed, we start planning from the current state. As such, planning and execution are interleaved.

- For each state s visited, many policies π are *evaluated* (partially)
- The quality of each π is approximated by averaging the expected reward of trajectories over S obtained by repeated *simulations* of $r(s, a, s')$.
- The chosen policy $\hat{\pi}$ is then selected and the action $\hat{\pi}(s)$ executed.

The question is: how to we do the repeated simulations? *Monte Carlo methods are by far the most widely-used approach.*

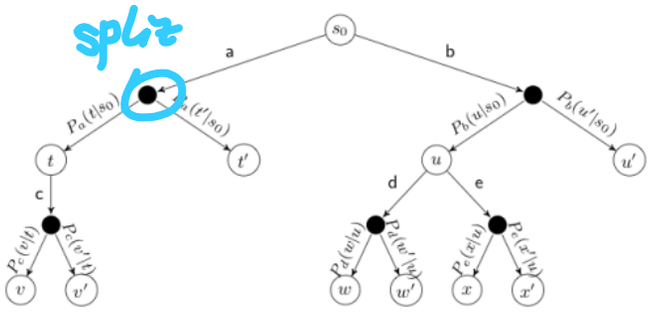
Monte Carlo

Monte Carlo Tree Search (MTCS) is a name for a *set* of algorithms all based around the same idea. Here, we will focus on using an algorithm for solving single-agent MDPs online.

Monte Carlo is an area within Monaco (small principality on the French riviera), which is best known for its extravagant casinos. As gambling and casinos are largely associated with chance, methods for solving MDPs online are often called *Monte Carlo* methods, because they use *randomness* to search the action space.

Foundation: MDPs as ExpectiMax Trees

To get the idea of MCTS, we note that MDPs can be represented as trees (or graphs), called *ExpectiMax* trees:



The letters a-e represent actions, and letters s-x represent states. White nodes are state nodes, and the small black nodes represent the probabilistic uncertainty: the 'environment' choosing which outcome from an action happens, based on the transition function.

Monte Carlo Tree Search – Overview

The algorithm is online, which means the action selection is interleaved with action execution. Thus, MCTS is invoked every time an agent visits a new state.

MCTS has the following fundamental features:

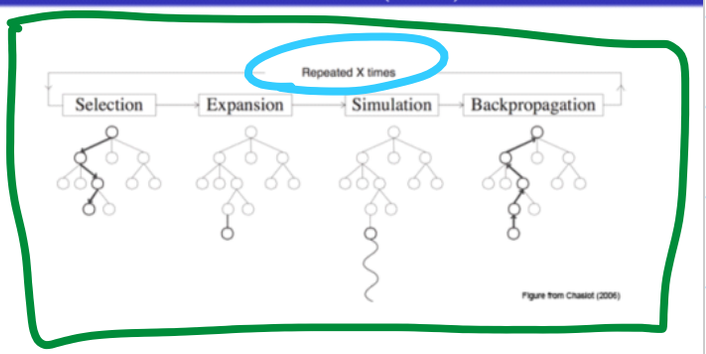
1. The value $V(s)$ for each is approximated using *random simulation*.
2. An ExpectiMax *search tree* is built incrementally.
3. The search terminates when some pre-defined computational budget is used up, such as a time limit or the number of expanded nodes;—therefore, it is an *anytime algorithm*, as it can be terminated at any time and it still provides an answer.
4. The best performing action is returned.
   - → This is complete if there are *no* dead–ends.
   - → This is optimal if an entire search can be performed (although this is unusual—if the problem is that small we should just use value/policy iteration).

The Framework: Monte Carlo Tree Search (MCTS)

MCTS builds up an MDP tree using simulation. The evaluated states are stored in a search tree. The set of evaluated states is *incrementally* built be iterating over the following four steps:

- *Select*: Select a single node in the tree that is *not fully expanded*. By this, we mean at least one of its children is not yet explored.
- *Expand*: Expand this node by applying one available action (as defined by the MDP) from the node.
- *Simulation*: From one of the new nodes, perform a complete random simulation of the MDP to a terminating state. This might typically assume that the search tree is finite, however versions for infinitely large trees exist in which we just execute for some time and then estimate the outcome.
- *Backpropagate*: Finally, the value of the node is *backpropagated* to the root node, updating the value of each ancestor node on the way using expected value.

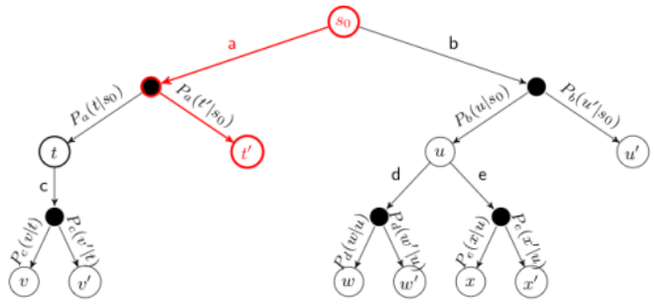
The Framework: Monte Carlo Tree Search (MCTS)



From: Chaslot, Guillaume, Sander Bakkes, Istvan Szita, and Pieter Spronck. *Monte-Carlo Tree Search: A New Framework for Game AI*. in *AIIDE*. 2008.
`https://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf`

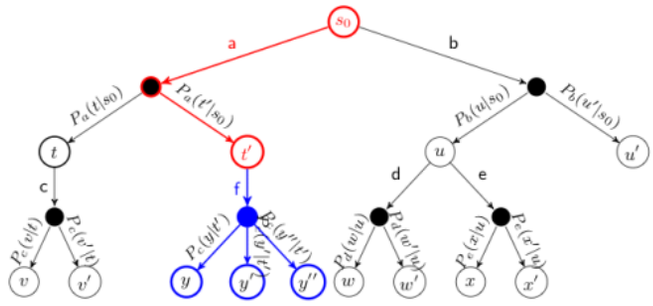
Monte Carlo Tree Search: Selection

Start at the root node, and successively select a child until we reach a node that is not fully expanded.



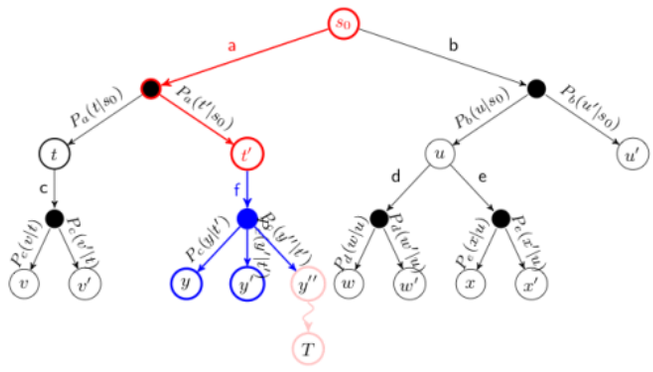
Monte Carlo Tree Search: Expansion

Unless the node we end up at is a terminating state, expand the children of the selected node by choosing an action and creating new nodes using the action outcomes.



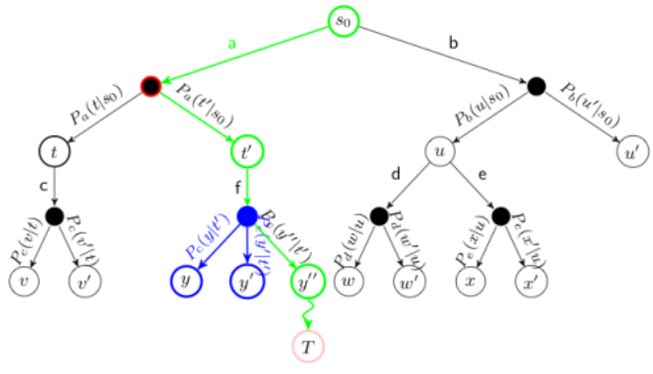
Monte Carlo Tree Search: Simulation

Choose one of the new nodes and *perform a random simulation of the MDP to the terminating state*:



Monte Carlo Tree Search: Backpropagation

Given the reward r at the terminating state, *backpropagate* the reward to calculate the value $V(s)$ at each state along the path.



Monte Carlo Tree Search: Algorithm

Input: MDP M, with initial state s_0 and time limit T.

```
function MctsSearch M = ⟨S, s_0, A, P_a(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s_0, a);
```

→ Each node stores an estimate of the value of the state, $V(s)$, for its state, the number of times the state has been visited, and a pointer to their parent node.

Monte Carlo Tree Search: Algorithm

Input: MDP M, with initial state s_0 and time limit T.

```
function MctsSearch M = ⟨S, s_0, A, P_a(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s_0, a);
```

Select($root$)

→ Recursively selects the next node using some probabilistic policy (we'll see more of this in 'Multi-Armed Bandits' section later), until we reach a node that is not fully expanded

→ At each choice point, selects one of the edges in the tree.

→ Then, using $P_a(s|s')$, select the outcome for that action. Thus, our *simulation* follows the probability transitions of the underlying model.

Monte Carlo Tree Search: Algorithm

Input: MDP M, with initial state s_0 and time limit T.

```
function MctsSearch M = ⟨S, s_0, A, P_a(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s_0, a);
```

Expand($expand\_node$)

→ Take the selected node and randomly select an action that can be applied in that state *and has not been selected previously in that state*.

→ Expand all possible outcomes nodes for that action.

→ Check if the generated nodes are already in tree. If not in the tree, *add* these nodes to the tree.

Note: $P_a(s|s')$ is *stochastic*, so several visits (in theory an infinite number) may be necessary to generate all successors.

Monte Carlo Tree Search: Algorithm

Input: MDP M, with initial state s_0 and time limit T.

```
function MctsSearch M = ⟨S, s_0, A, P_a(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s_0, a);
```

Simulate($child$)

- → Perform a random simulation of the MDP until we reach a terminating state. That is, at each choice point, randomly select an enable action from the MDP, and use transition probabilities $P_a(s'|s)$ to choose an outcome for each action.
- → We can use non-random simulation as well by following some heuristic, but we will not look at this in these notes.
- → $reward$ is the reward obtained over the entire simulation.
- → To avoid memory explosion, we *discard* all nodes generated from the simulation. In any non-trivial search, we are unlikely to ever need them again.

Monte Carlo Tree Search: Algorithm

Input: MDP M, with initial state s_0 and time limit T.

```
function MctsSearch M = ⟨S, s_0, A, P_a(s'|s), R(s)⟩ returns action a
  while current_time < T do
    expand_node := Select(root);
    children := Expand(expand_node);
    child := Choose(children); – choose a child to simulate
    reward := Simulate(child); – simulate from child
    Backup(expand_node, reward);
  return argmax_a Q(s_0, a);
```

Backup($expand\_node$,$reward$)

- → The reward from the simulation is backpropagated from the expanded node to its ancestors recursively.
- → We must not forget the *discount factor*!
- → For each state s, get the expected value of all actions from that node:

$$V(s) := \max_{a \in A(s)} \sum_{s' \in children} P_a(s'|s)\,[r(s,a,s') + \gamma\, V(s')]$$

Look familiar?!

This is why the tree is called an ExpectiMax tree: we maximise the expected return, and this calculation is done over two layers. The summation ($\sum_{s' \in S} \ldots$) is calculating the value of the small black nodes in the tree, while the maximisation ($\max_{a \in A(s)}$) calculates the value of the large white nodes (the state nodes).

Monte Carlo Tree Search: Execution

Once we have run out of computational time, we select the action that maximises are expected return, which is simply the one with the highest Q-value from our simulations:

$$\operatorname*{argmax}_a Q(s_0, a)$$

which is just

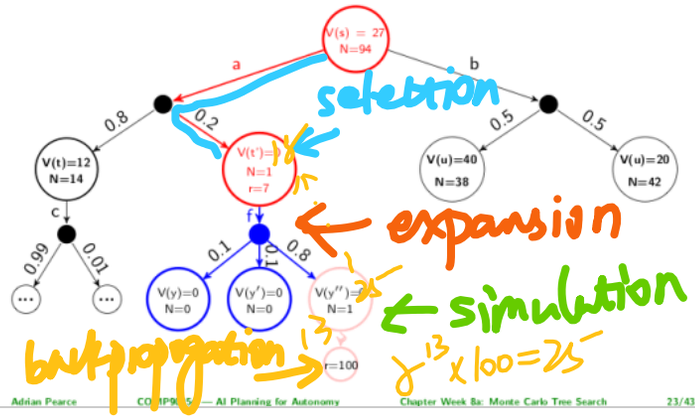
$$\operatorname*{argmax}_a \sum_{s' \in A(s_0)} P_a(s'|s_0)\,[r(s_0,a,s') + \gamma\, V(s')]$$

We execute that action and wait to see which outcome occurs for the action. Once we see the outcome, which we will call s', we start the process all over again, except with $s_0 := s'$.

Importantly, we can *keep* the sub-tree from state s', as we already have done simulations from that state. We discard the rest of the tree (all child of s_0 other than the chosen action) and incrementally build from s'.

Example (after the simulation step)

Assume $\gamma = 0.9$, $r = X$ represents reward X received at a state, N is the number of times the state has been visited, and the length of the simulation is 13. After the simulation step, but before backpropagation, our tree would look like this:

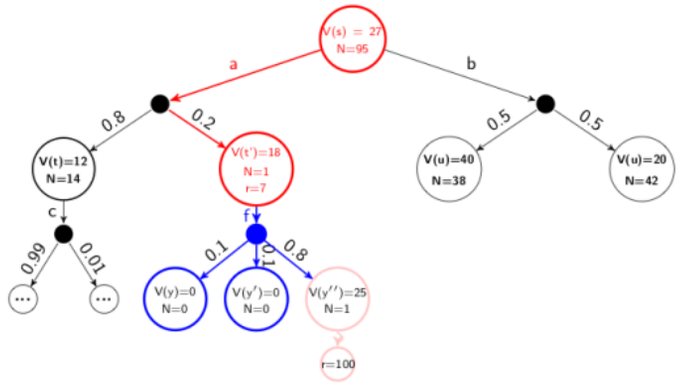


Example (the backpropagation step)

$$
\begin{aligned}
V(y'') &= \max_{a \in A} \sum_{s' \in children(y'')} P_a(s'|y'') \; [r(y'', a, s') + \gamma \; V(s')] \\
&= \gamma^{13} \times 100 \text{ (simulation is 13 steps long and receives reward of 100)} \\
&\approx 25 \\
V(t') &= \max_{a \in \{f\}} \sum_{s' \in children(t')} P_a(s'|t') \; [r(t', a, s') + \gamma \; V(s')] \\
&= 0.1(0+0) \; + \; 0.1(0+0) \; + \; 0.8(0 + 0.9 \times 25) \\
&= 18 \\
V(s) &= \max_{a \in \{a,b\}} \sum_{s' \in children(s)} P_a(s'|s) \; [r(s, a, s') + \gamma \; V(s')] \\
&= \max(0.8(0 + 0.9 \times 12) + 0.2(7 + 0.9 \times 18), \quad \text{(action a)} \\
&\qquad 0.5(0 + 0.9 \times 40) + 0.5(0 + 0.9 \times 20) \quad \text{(action b)} \\
&= \max(8.64 + 4.62, \; 18 + 9) \\
&= 27
\end{aligned}
$$

Example (after the backpropagation step)

The value of $V(s)$ does not change because action b still returns the maximum discounted future reward.



Informed Search

There is one key question that we need to answer:

How do we select the next node to expand?

It turns out that this selection makes a big difference on the performance of MCTS.

Multi-Armed Bandit: Informal Definition

The selection of nodes can be considered an instance of the *Multi-armed bandit* problem. This problem is defined as follows:

> *Imagine that you have N number of slot machines (or poker machines in Australia), which are sometimes called one-armed bandits. Over time, each bandit pays a random reward from an unknown probability distrbution. Some bandits pay higher rewards than others. The goal is to maximize the sum of the rewards of a sequence of lever pulls of the machine.*

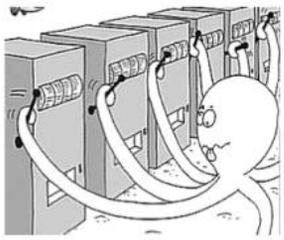


Image courtesy of Mathworks blog: `https://blogs.mathworks.com/loren/2016/10/10/multi-armed-bandit-problem-and-exploration-vs-exploitation-trade-off/`

Multi-Armed Bandit: Formal Definition

An N-armed bandit is defined by a set of *random variables* $X_{i,k}$ where

- $1 \leq i \leq N$, such that i is the *arm* of the bandit; and
- k the index of the *play* of arm i.

Successive plays $X_{i,1}, X_{j,2}, X_{k,3} \ldots$ are assumed to be independently distributed according to an *unknown* law. That is, we do not know the probability distributions of the random variables.

Intuition: actions a applicable on s are the "arms of the bandit", and $Q(s,a)$ corresponds to the random variables $X_{i,n}$.

Flat Monte Carlo (FMC) a.k.a Uniform Sampling

Given that we do not know the distributions, a simple strategy is simply to select the arm given a uniform distribution; that is, select each arm with the same probability. This is just uniform sampling.

Then, the Q-value for an action a in a given state s can be approximated using the following formula:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{t=1}^{N(s)} \mathbb{I}_t(s,a) r_t$$

- $N(s,a)$ is the number of times a executed in s.
- $N(s)$ is the number of times s is visited.
- r_t is the *reward* obtained by the t-th simulation from s.
- $\mathbb{I}_t(s,a)$ is 1 if a was selected on the t-th simulation from s, and is 0 otherwise

→ FMC suffices to achieve *world champion level* play on Bridge (Ginsberg, 01) and Scrabble (Sheppard, 02).

But what is the issue? Sampling Time is wasted equally in all actions using the uniform distribution. Why not focus also on the *most promising actions* given the rewards we have received so far.

Exploration vs. Exploitation

What we want is to play only the good actions; so just keep playing the actions that have given us the best reward so far. However, our selection is randomised, so what if we just haven't sampled the best action enough times? Thus, we want strategies that *exploit* what we think are the best actions so far, but still *explore* other actions.

But how much should we exploit and how much should we explore? This is known as the *exploration vs. exploitation dilemma*.

It is driven by the *The Fear of Missing Out* (FOMO)

The Fear Of Missing Out

We seek policies π that *minimise regret*.

(Pseudo)–Regret

$$\mathcal{R}_{N(s),b} = Q(\pi^*(s),s)N(s) - \mathbb{E}[\sum_{t}^{N(s)} Q(b,s)\mathbb{I}_t(s,b)]$$

- $Q(\pi^*(s),s)$ is the Q-value for the (unknown) optimal policy $\pi^*(s)$,
- $N(s)$ is the number of visits to state s,
- $\mathbb{I}_i(s,a)$ is 1 if a was selected on the i-th visit from s, and 0 otherwise,
- **Important**: $\mathbb{E}[\sum_{t}^{N(s)} Q(b,s)\mathbb{I}_t(s,b)] > 0$ for every b.

Informally: If I play arm b, my regret is the *best possible expected reward* minus the *expected reward of playing* b. If I play arm a (the best arm), my regret is 0.

Regret is thus the *expected loss* due to not doing the best action.

→ In multi-armed bandit algorithms, exploration is *literally* driven by FOMO.

Solutions that aim to minimise regret

ϵ-greedy: ϵ is a number in [0,1]. Each time we need to choose an arm, we choose a random arm with probability ϵ, and choose the arm with max $Q(s,a)$ with probability $1-\epsilon$. Typically, values of ϵ around 0.05-0.1 work well.

ϵ-decreasing: The same as ϵ-greedy, ϵ decreases over time. A parameter α between [0,1] specifies the *decay*, such that $\epsilon := \epsilon.\alpha$ after each action is chosen.

Softmax: This is *probability matching strategy*, which means that the probability of each action being chosen is dependent on its Q-value so far. Formally:

$$\frac{e^{Q(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q(s,b)/\tau}}$$

in which τ is the *temperature*, a positive number that dictates how much of an influence the past data has on the decision.

Upper Confidence Bounds (UCB1)

A highly effective (especially in terms of MCTS) multi-armed bandit strategy is the *Upper Confidence Bounds* (UCB1) strategy.

UCB1 policy $\pi(s)$

$$\pi(s) := \underset{a \in A(s)}{\operatorname{argmax}}\ Q(s,a) + \sqrt{\frac{2 \ln N(s)}{N(s,a)}}$$

$Q(s,a)$ is the estimated Q-value.
$N(s)$ is the number of times s has been visited.
$N(s,a)$ is the number of times times a has been executed in s.

→The left–hand side encourages exploitation: the Q-value is high for actions that have had a high reward.
→The right–hand side encourages exploration: it is high for actions that have been explored less.

Upper Confidence Trees (UCT)

UCT = MCTS + UCB1

Kocsis & Szepesvári (2006) were the first to treat the selection of nodes to expand in MCTS as a multi-armed bandit problem.

UCT exploration policy

$$\pi(s) := \underset{a \in A(s)}{\operatorname{argmax}}\ Q(s,a) + 2C_p\sqrt{\frac{2 \ln N(s)}{N(s,a)}}$$

$C_p > 0$ is the exploration constant, which determines can be increased to encourage more exploration, and decreased to encourage less exploration. Ties are broken randomly.

→ if $Q(s,a) \in [0,1]$ **and** $C_p = \frac{1}{\sqrt{2}}$ **then** in two-player adversarial games, UCT converges to the well-known Minimax algorithm (if you don't know what Minimax is, ignore this for now and we'll mention it later in the subject).

What if we do not know $P_a(s' \mid s)$?

In the coming lectures, we will look more at situations in which we do not know $P_a(s' \mid s)$, but it is important to note that we can use MCTS even if we do not know our transition probabilities or our reward function, provided that we can *simulate* them; e.g. using a code-based simulator. This leads to a new approach based on a straightforward modification:

1. *Selection* is as before.
2. In the *expansion* step, instead of expanding all child nodes of an action, we run the simulation forward one step, which will choose an outcome according to $P(s' \mid s)$ (provided, or course, that the simulator is accurate).
3. We then simulate as before, and we learn the rewards when we receive them.
4. In the *backpropagation* step, instead of using the Bellman equation to calculate the expected return, we simply use the *average* return. If we simulate each step enough times, the average will converge to the expected return.

The disadvantage of this approach is that we have to do repeated simulations for the average to converge, whereas when we know $P_a(s' \mid s)$, we need only expand an action once to know its immediate effect.
The advantage is that it is more general: as long as we have a simulator for our problem, we can apply it – we do not need an explicit model of the problem. For many problem, simulators are easier to produce than problems.

Applications of MCTS with UCB tree policies

Games:

- Go: MoGo (2006), Fuego (2009), ..., Alpha Go(2010–2016)
- Board Games: Havannah, Y, Cataan, Othello, Arimaa...
- Video Games: Atari 2600

Not Games:

- Computer Security: Attack tree generation & Penetration testing
- Deep Learning: Automated "performance tuning" of Neural Nets and Feature Selection
- Operations Research: Optimising bus schedules, inventory stock management, logistics operations,...

Why does it work so well (sometimes)?

MCTS addresses exploitation vs. exploration comprehensively.

UCT is *systematic*:

- Policy evaluation is *exhaustive* up to a certain depth.
- Exploration aims at *minimising regret* (or FOMO).

Watch it playing Mario Bros.
(`https://www.youtube.com/watch?v=HRiEUUC9TUA`).

Where it does not do so well...: Atari 2600 game Freeway
(`https://www.youtube.com/watch?v=YVbTbMO4rtM`)

It fails for Freeway because the character does not receive a reward until it reaches the other side of the road, so UCT has no feedback to go on.

- Compare this with Simulated Best-First Width Search (BFWS)/IW algorithm from classical planning, which does much better as it does not rely on *rewards* but instead on *novelty* (see lecture: 5. Width Based Planning)

Value/policy iteration vs. MCTS

Often the set of states reachable from the initial state s_0 using an optimal policy is much small that the set of total states. In this regard, value iteration and policy iteration are exhaustive: they calculate behaviour from states that may never be encountered if we know the initial state of the problem.

MCTS (and other search methods) methods thus can be used by just taking samples starting at s_0. However, the result is not as general as using value/policy iteration: the resulting solution *will work only from the known initial state* s_0 or any state reachable from s_0 using actions defined in the model. Whereas value/policy iteration methods work from any state.

Value/policy iteration vs. MCTS

	Value/policy iteration	**MCTS**
Cost	Higher cost (exhaustive)	Lower cost (does not solve for entire state space)
Coverage/ Robustness	Higher (works from any state)	Low (works only from initial state or state reachable from initial state)

This is important: value/policy iteration are thus more expensive, however, for an agent operating in its environment, we only solve exhaustively once, and we can use the resulting policy many times no matter state we are
For MCTS, we need to solve *online* each time we encounter a state we have not considered before.

Summary

- Monte Carlo Tree Search (MCTS) is an anytime search algorithm, especially good for stochastic domains, such as MDPs.
  - Smart selection strategies are *crucial* for good performance.
- Upper Confidence Bounds (UCB1) for Multi-Armed Bandits makes a good selection policy.
  - UCB1 (with slight modifications) balances exploitation and exploration remarkable well.
  - The Fear Of Missing Out is an *excellent* motivator for exploration.
- UCT is the combination of MCTS and UCB1, and is an very successful algorithm for many problems.
  - Yet it has obvious shortcomings, e.g. relies on sufficient reward signal to work, cannot handle dead-ends (at least in standard form), etc.
  - There are alternatives to FOMO to motivate exploration, such as ϵ-greedy and softmax.

Partially Observable MDPs

MDPs assume that the agent always knows exactly what state it is in — the problem is fully-observable.

However, this is not valid for many tasks; e.g. an unmanned aerial vehicle searching in a earthquake zone for survivors will by definition not know the location of survivors; a card-player agent will not know the cards its opponent holds; etc.

Partially-observable MDPs (POMDPs) relax the assumption of full-observability.

A POMDP is defined as:

- states $s \in S$
- set of goal states $G \subseteq S$
- 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
- reward function $r(s, a, s')$
- a **sensor model** given by probabilities $P_a(o|s)$, $o \in Obs$

Solving POMDPs (an intuitive overview)

Solving POMDPs is very similar to solving MDPs. In fact, the same algorithms apply. The only difference is that we case the POMDP problem as a standard MDP problem with a new state space: each state is a **probability distribution** over the set S. Thus, each state of the POMDP is a **belief state**, which defined the probability of being in each state S.

Like MDPs, solutions are policies that map belief states into actions.

Optimal policies minimise the expected reward to go from b_0 to G.

We will not cover this in detail in these notes. However, POMDPs are clearly a generalisation of MDPs, and they have had a much larger impact on planning for autonomy than standard MDPs.

Summary: MDPs

We covered Markov Decision Processes (MDPs). They differ from classical planning in that actions can have more than one possible outcome. Each outcome has an associated probability.

There are two solutions for exhaustively calculating the optimal policy: value iteration and policy iteration. These are both based on dynamic programming – specifically, they use the Bellman equations to iteratively improve on a non-optimal solution.

Heuristic search can also be used, but does not produce solutions that are as general – the work only for states that are reachable from the initial state of the search.

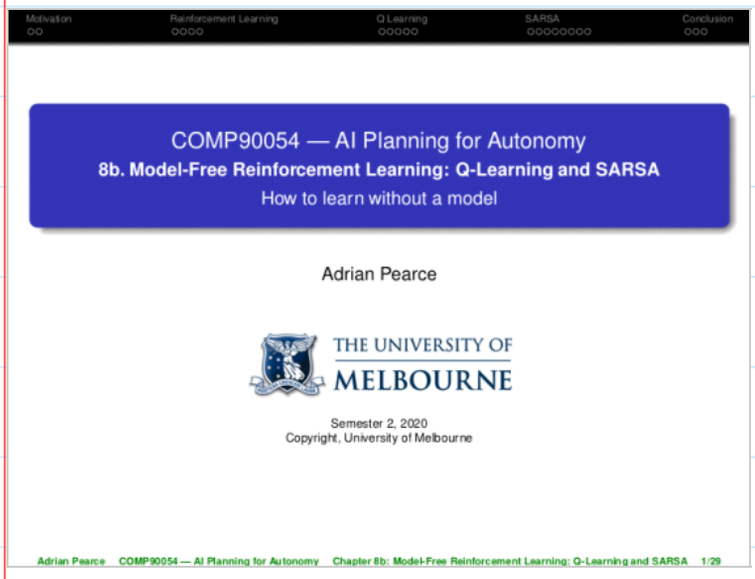
Partially-observable MDPs generalise MDPs by admitting descriptions in which the environment is not fully observable. Techniques for solving these are the same as MDPs, but just over a larger search space.

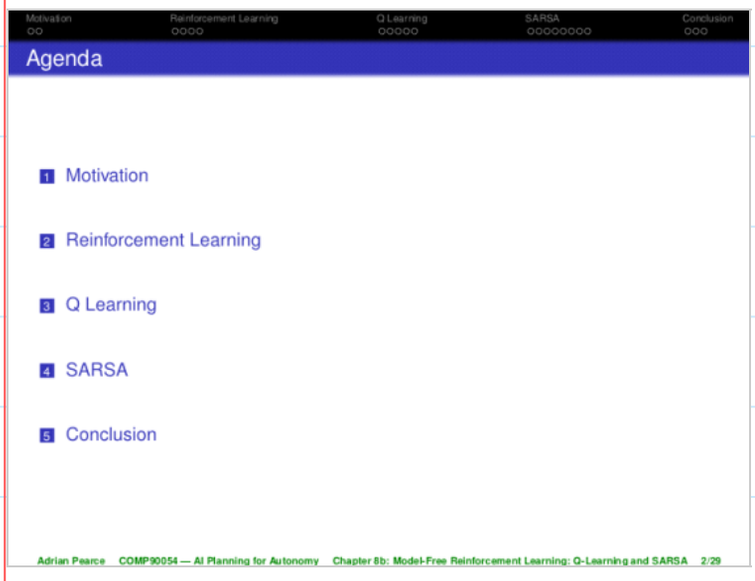
What's next? How to *learn* the probabilities over the action outcomes using *reinforcement learning*.

Chapter 10

2020年9月24日 星期四 14:40

intro8b-pr
e-handout





Learning Outcomes

1. Identify situations in which model-free reinforcement learning is a suitable solution for an MDP
2. Explain how model-free planning differs from model-based planning
3. Apply Q-learning and SARSA to solve small-scale MDP problems manually and program Q-learning and SARSA algorithms to solve medium-scale MDP problems automatically
4. Compare and contrast off-policy reinforcement learning with on-policy reinforcement learning

Planning and Learning

So far, the this subject, we have at looked blind/heuristic search and value/policy iteration.

- Search and value/policy iteration are what are called as *model-based* techniques. This means that we need to know the model: in particular, we have access to $P_a(s' \mid s)$ and $r(s, a, s')$.
- Q-learning and SARSA (discussed in this lecture) are *model-free* techniques. This means that we do NOT know the $P_a(s' \mid s)$ and $r(s, a, s')$.

Key Question: How can we calculate a policy if we don't know the transitions and the rewards?!

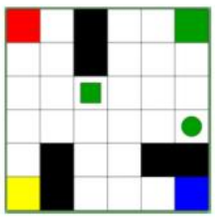
Answer: We can *learn through experience* by trying actions and seeing what the results is, making this machine learning problem.

- Importantly, in model-free reinforcement learning, we do NOT try to learn $P_a(s' \mid s)$ or $r(s, a, s')$ — we learn a policy *directly*.
- There is something in between model-based and model-free: simulation-based techniques. In this cases, we have a model as a *simulator*, so we can *simulate* $P_a(s' \mid s)$ and $r(s, a, s')$ and learn a policy with a model-free technique.

experience → policy

Example: The Mystery Game

https://programmingheroes.blogspot.com/2016/02/udacity-reinforcement-learning-mystery-game.html



From this website (in Spanish, Chrome offered to translate into English when I visited): The aim of this game is to experiment how computers learn. Press keys from 1 to 6 to do actions. You need to learn what effects/outcomes the actions produce and how to win the game.

Some rewards values appears when you do the things very well or very bad. When you finish the game the phrase "You Win :)" appears in the board. Good luck!

Example: Mystery Game (continued)

What was the process you took?

- What did you learn?
- What assumptions did you use?

→ Imagine how hard it is for a computer that doesn't have any assumptions or intuition!

Approaches to AI Planning and Reinforcement Learning

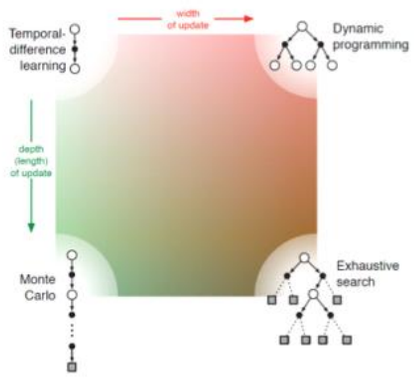


Figure from Sutton & Barto (read Section 8.13 for description of this figure)

Reinforcement Learning: The Basics

There are many different models of reinforcement learning, all with the same basis:

We execute many different *episodes* of the problem we want to solve, and from that we learn a *policy*.

- During learning, we try to learn the value of applying particular actions in particular states.
- During each episode, we need to execute some actions. After each action, we get a reward (which may be 0) and we can see the new state.

From this, we *reinforce* our estimates of applying the previous action in the previous state.

- We terminate when: (1) we run out of training time; (2) we think our policy has converged to the optimal policy (for each new episode we see no improvement); or (3) our policy is 'good enough' (when for each new episode we see only minimal improvement).

Q-Learning

Q-Learning is perhaps the simplest of reinforcement learning methods, and is based on how animals learn from their environment. The intuition is quite straightforward.

Maintain a Q-function that records $Q(s, a)$ for every state-action pair. At each step: (1) choose an action using a multi-armed bandit algorithm; (2) apply that action and receive the reward; and (3) update $Q(s, a)$ based on that reward. Repeat over a number of episodes until . . . when?

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Figure: Q-Learning Algorithm (from Sutton and Barto)

Updating the Q-function

Updating the Q-function (line 7) is where the learning happens:

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{old value}} + \overbrace{\alpha}^{\text{learning rate}} \cdot [\underbrace{r}_{\text{reward}} + \overbrace{\gamma}^{\text{discount factor}} \cdot \underbrace{\max_{a'} Q(s',a')}_{\text{estimate of optimal future value}} \overbrace{-Q(s,a)}^{\text{do not count extra } Q(s,a)}]$$

A higher learning rate α will weight more recent information higher than older information ($Q(s, a)$).

Note that we estimate the future value using $\max_{a'} Q(s', a')$, which means it *ignores* the action chosen by the policy, and instead updates based on the estimate of the best action for the updated state s'.

This is known as *off policy* learning – more on this later.

Q-functions using Q-Tables

Q-tables are the simplest way to maintain a Q-function. They are a table with an entry for every $Q(s, a)$. Thus, like value functions in value iteration, they do not scale to large state-spaces. (More on scaling in the next lecture).

Initially					After some training				
State	Action				State	Action			
	North	South	East	West		North	South	East	West
(0,0)	0	0	0	0	(0,0)	0.53	0.36	0.36	0.21
(0,1)	0	0	0	0	(0,1)	0.61	0.27	0.23	0.23
. . .					. . .				
(2,2)	0	0	0	0	(2,2)	0.79	0.72	0.90	0.72
(2,3)	0	0	0	0	(2,3)	0.90	0.78	0.99	0.81

Q-learning: Example

After some training

State	Action			
	North	South	East	West
(0,0)	0.53	0.36	0.36	0.21
(0,1)	0.61	0.27	0.23	0.23
...				
(2,2)	0.79	0.72	0.90	0.72
(2,3)	0.90	0.78	0.99	0.81

In state (2,2), the action 'North' is chosen and executed successfully, which would return to state (2,2) there is no cell above (2,2). Using the Q-table above, we would update the Q-value as follows:

$$\begin{aligned} Q((2,2),N) &\leftarrow Q((2,2),N) + \alpha[r + \gamma \max_{a'} Q((2,2),a') - Q((2,2),N)] \\ &\leftarrow 0.79 + 0.1[0 + 0.9 \cdot Q((2,2),East) - Q((2,2),N)] \\ &\leftarrow 0.79 + 0.1[0 + 0.9 \cdot 0.90 - 0.79] \\ &\leftarrow 0.792 \end{aligned}$$

Using Q-functions

We iterate over as many episodes as possible, or until each episode hardly improves our Q-values. This gives us a (close to) optimal Q-function.

Once we have such a Q-function, we stop exploring and just exploit. We use *policy extraction*, which is exactly as we do for value iteration:

$$\pi(s) = \underset{a \in A(s)}{\operatorname{argmax}}\, Q(s,a)$$

SARSA: On-Policy Reinforcement Learning

SARSA = State-action-reward-state-action

On-Policy: Instead of using $Q(s', a')$ for the best estimated future state during update, on-policy uses the actual next action to update:

- On-policy learning estimates $Q^{\pi}(s,a)$ state action pairs, for the current behaviour policy π, whereas off-policy learning estimates the policy independent of the current behaviour.

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

On-Policy: Uses the action chosen by the policy for the update!

Q-learning vs. SARSA

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Figure: Q-Learning Algorithm

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Figure: Sarsa Algorithm

Off-Policy: Ignores the action chosen by the policy, uses the best action $\text{argmax}_{a'} \mathcal{Q}(s', a')$ for the update!

On-Policy SARSA learns action values relative to the policy it follows, while Off-Policy Q-Learning does it relative to the greedy policy.

On-policy vs. off-policy: What is the difference here?

The difference is all in how the update happens in the loop body.

Q-learning: (1) selects an action a; (2) takes that actions and observes the reward & next state s'; and (3) updates *optimistically* by assuming the future reward is $\max_{a'} Q(s', a')$ – that is, it assumes that future behaviour will be optimal (according to its policy).

SARSA: (1) selects action a' for the *next* loop iteration; (2) in the next iteration, takes that action and observes the reward & next state s'; (3) only then chooses a' for the next iteration; and (4) updates using the estimate for the actual next action chosen – which may not be the greediest one (*e.g. it could be selected so that it can explore*).

SARSA: Example

After some training

State	Action			
	North	South	East	West
(0,0)	0.53	0.36	0.36	0.21
(0,1)	0.61	0.27	0.23	0.23
...				
(2,2)	0.79	0.72	0.90	0.72
(2,3)	0.90	0.78	0.99	0.81

In state (2,2), the action 'North' is chosen and executed successfully, which would return to state (2,2) there is no cell above (2,2). The next selected action is West. Using the Q-table above, we would update the Q-value using SARSA as follows:

$$\begin{aligned} Q((2,2),N) &\leftarrow Q((2,2),N) + \alpha[r + \gamma Q((2,2),W) - Q((2,2),N)] \\ &\leftarrow 0.79 + 0.1[0 + 0.9 \cdot Q((2,2),W) - Q((2,2),N)] \\ &\leftarrow 0.79 + 0.1[0 + 0.9 \cdot 0.72 - 0.79] \\ &\leftarrow 0.7758 \end{aligned}$$

On-policy vs. off-policy: Who cares??

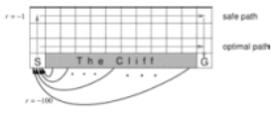
So what difference does this really make? There are two main differences:

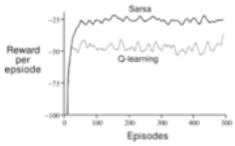
- Q-learning will converge to the optimal policy irrelevant of the policy followed, because it is *off-policy*: it uses the greedy reward estimate in its update rather than following the policy such as ϵ-greedy). Using a random policy, Q-learning will still converge to the optimal policy, but SARSA will not (necessarily).
- Q-learning learns an optimal policy, but this can be 'unsafe' or risky *during training*.

SARSA vs. Q-learning: Example

Consider the grid below. S is the start and state G receives a reward of 100. Falling off the cliff receives a reward of -100. Going to the top row receives a -1 reward. Actions are deterministic, but they are not known before learning.

Q-learning leads the optimal path along (along the edge of the cliff), but will fall off sometimes due to the ϵ-greedy action selection. SARSA learns the safe path because it is on-policy, and considers the action select method when learning. Here is a graph showing the reward per trial for both Sarsa and Q-Learning:





Rewards during training

SARSA receives a higher average reward *per trial* than Q-Learning, because it falls off the cliff less in later episodes. However, Q-learning learns the *optimal* policy.

Demo of the cliff example: `https://studywolf.wordpress.com/2013/07/01/reinforcement-learning-sarsa-vs-q-learning/`

On-policy vs. off policy: Why do we have both?

Imagine a reinforcement learning agent that manages resources for a cloud-based platform and we have no prior data to inform a policy.

- On-policy learning is more appropriate when we want to optimise the behaviour of an agent who learns *while operating in its environment*.
  We would need to operate our cloud platform to get data. As such, if the average reward *per trial* is better using on-policy, this would give us better overall outcomes than off-policy learning, because the 'trials' are not practice – they actually influence how much money we make.
- Off-policy learning is more appropriate when we have the luxury of training our agent offline before it is put into operation.

  If we could run our reinforcement learning algorithm in a simulated environment before deploying (and we had reason to believe that simulated environment was accurate), off-policy learning would be better because its optimal policy could be followed.

In short: use on-policy reinforcement learning for online learning, and off-policy learning for offline learning.

Example: if we used reinforcement learning for traffic light optimisation, we would use on-policy reinforcement learning, because we could not train it before hand given that the policy influences the behaviour of drivers.

Q-learning Examples in Action

In action: solving the cliff example using Q-learning with ϵ-greedy:

https://www.youtube.com/watch?v=ppALjH0kYPE

The source code for this:

https://github.com/alecKarfonta/Gridworld

Worked example: a complete worked example of using Q-learning to calculate the optimal path:

http://www.mnemstudio.org/path-finding-q-learning-tutorial.htm

Summary

If we know the MDP:

- **Offline**: Value Iteration, Policy Iteration,
- **Online**: Monte Carlo Search Tree and friends.

If we do *not* know MDP:

- **Offline**: Reinforcement Learning
- **Online**: Monte Carlo Tree Search and friends.

Once you've got your pacman Q-learning working in python (optional assessment and bonus exercise in the workshop), you can test it on all the environments on OpenAI, a Toolkit for developing and testing reinforcement learning algorithms:

https://gym.openai.com/

Applications of Reinforcement Learning

- Checkers (Samuel, 1959)
  first use of RL in an interesting real game
- (Inverted) Helicopter Flight (Ng et al. 2004)
  better than any human
- Computer Go (AlphaGo 2016)
  AlphaGo beats Go world champion Lee Sedol 4:1
- Atari 2600 Games (DQN & Blob-PROST 2015)
  human-level performance on half of 50+ games
- Robocup Soccer Teams (Stone & Veloso, Reidmiller et al.)
  World's best player of simulated soccer, 1999; Runner-up 2000
- Inventory Management (Van Roy, Bertsekas, Lee & Tsitsiklis)
  10-15% improvement over industry standard methods
- Dynamic Channel Assignment (Singh & Bertsekas, Nie & Haykin)
  World's best assigner of radio channels to mobile telephone calls
- Elevator Control (Crites & Barto)
  (Probably) world's best down-peak elevator controller
- Many Robots
  navigation, bi-pedal walking, grasping, switching between skills, ...
- TD-Gammon and Jellyfish (Tesauro, Dahl)
  World's best backgammon player. Grandmaster level

Reading

- Chapter 6: Temporal-Difference Learning of *Reinforcement Learning: An Introduction (Second Edition)*, 2020 [*Sutton and Barto*]

  Available at:

  `http://www.incompleteideas.net/book/RLbook2020.pdf`

  Content: Great entry level book to Reinforcement Level written by the founders of the field.

- *Slides about Approximate Q-learning for PacMan*

  Available at:

  `https://www.cs.swarthmore.edu/~bryce/cs63/s16/slides/3-25_approximate_Q-learning.pdf`

  Content: Great technique if you want to use reinforcement learning for the competition!

- *Deep Q-learning for Atari*

  Available at:

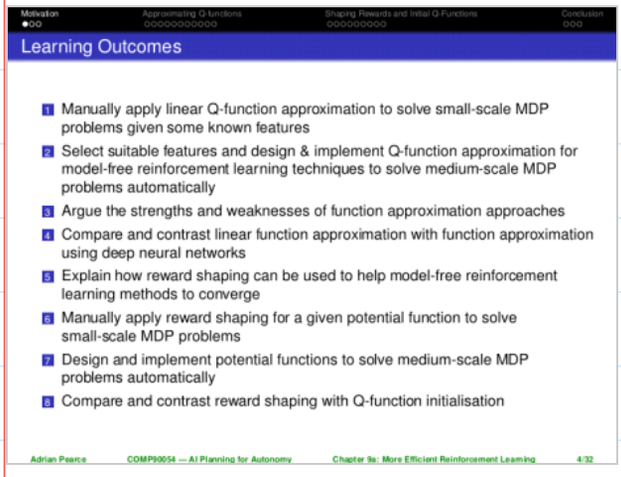
  `http://www.davidqiu.com:8888/research/nature14236.pdf`

  Content: Convolutional Neural Networks (NN) to estimate $\mathcal{Q}(s, a)$. The input for the NN is the state, and the output is the esimated reward for each action.

Chapter 11

2020年10月1日 星期四 14:46

intro09a-p
re-hando...



Reinforcement Learning – Some Weaknesses

In the previous lectures, we looked at fundamental temporal difference (TD) methods for reinforcement learning. As noted, these two methods have some weaknesses in this basic format:

1. Unlike Monte-Carlo methods, which reach a reward and then backpropagate this reward, TD methods use bootstrapping (they estimate the future discounted reward using $Q(s,a)$), which means that for problems with sparse rewards, it can take a long time to for rewards to propagate throughout a Q-function.
2. Both methods estimate a Q-function $Q(s,a)$, and the simplest way to model this is via a Q-table. However, this requires us to maintain a table of size $|A| \times |S|$, which is prohibitively large for any non-trivial problem.
3. Using a Q-table requires that we visit every reachable state many times and apply every action many times to get a good estimate of $Q(s,a)$. Thus, if we never visit a state s, we have no estimate of $Q(s,a)$, even if we have visited states that are very similar to s.
4. Rewards can be sparse, meaning that there are few state/actions that lead to non-zero rewards. This is problematic because initially, reinforcement learning algorithms behave entirely randomly and will struggle to find good rewards. Remember the Freeway demo from the previous lecture?

Reinforcement Learning – Some Improvements

To get around these limitations, we are going to look at three simple approaches that can improve temporal difference methods:

1. *n-step temporal difference learning*: Monte Carlo techniques execute entire traces and then backpropagate the reward, while basic TD methods only look at the reward in the next step, estimating the future wards. n-step methods instead look n steps ahead for the reward before updating the reward, and then estimate the remainder. *Next lecture!*
2. *Approximate methods*: Instead of calculating an exact Q-function, we approximate it using simple methods that both eliminate the need for a large Q-table (therefore the methods scale better), and also allowing use to provide reasonable estimates of $Q(s,a)$ *even if we have not applied action a in state s previously. This lecture!*
3. *Reward shaping and Q-Value Initialisation*: If rewards are sparse, we can modify/augment our reward function to reward behaviour that we think moves us closer to the solution, or we can guess the optimal Q-function and initial $Q(s,a)$ to be this. *This lecture!*

Problem: Large State Spaces

As discussed, the problem is that the size of the state space increases exponentially with every variable added. This causes two main issues:

1. Storing a value of $Q(s,a)$ for every s and a, such as in Q-tables, is infeasible.
2. Propagation of Q-values takes a long time.

Example: *Freeway*. Let's say there are 12 rows and about 40 columns (this underestimates because the cars move a few pixels at a time, not in columns), so about 480 different positions. There are two chickens, plus we need to store whether there is a car in each location (at the very least). This leads to:

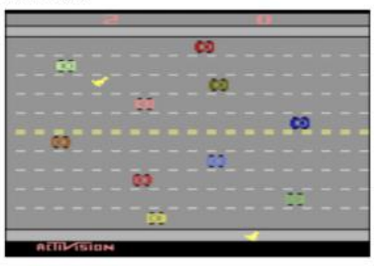
$$480^2 + 2^{480} = 3.12 \times 10^{144} \text{ states}$$

And therefore we would need a Q-table this size times the number of actions (Four actions I think: left, right, up, down?).

$|A|$

Feature Selection Exercise: Freeway

What would be some good features for the Freeway game to learn how to get the chicken across the freeway?



① feature function that returns a feature vector

$f(s,a) = \begin{pmatrix} f_1(s,a) \\ f_2(s,a) \\ \vdots \\ f_n(s,a) \end{pmatrix}$ → return one feature each function

n features in total

weight vector: $n \times |A|$ (normalize)

features, # actions

$f(s) = \begin{pmatrix} f_1(s) \\ f_2(s) \\ \vdots \\ f_n(s) \end{pmatrix}$ n

$f_{2a}(s,a) = \begin{pmatrix} f_{1a}(s,a) \\ f_{2a}(s,a) \\ f_{na}(s,a) \end{pmatrix}$ $n \times |A|$ a_1 a_1

Linear Function Approximation

The key idea is to *approximate* the Q-function using a linear combination of *features* and their weights. Instead of recording everything in detail, we think about what is

Linear Function Approximation

The key idea is to *approximate* the Q-function using a linear combination of *features* and their weights. Instead of recording everything in detail, we think about what is most important to know, and model that.

The overall process is:

1. For the states, consider what are the features that determine its representation.
2. During learning, perform updates based on the *weights* of *features* instead of states.
3. Estimate $Q(s,a)$ by summing the features and their weights.

Example: *Freeway*. Instead of recording the position of both players and whether there is a car in every position, just record how far away each player is from the other side of the road and how far away the *closest* car is in each row. This is probably enough. This requires just 14 features. In approximate Q-learning, we store features and weights, not states. What we need to learn is how important each feature is (its *weight*) for each action.

Approximate Q-function Representation

As noted, a Q-function is represented using the features and their weights, instead of a Q-table.

To represent this, we have two vectors:

1. A *feature vector*, $f(s,a)$, which is a vector of $n \cdot |A|$ different functions, where n is the number of state features and $|A|$ the number of actions. Each function extracts the value of a feature for state-action pair (s,a). We say $f_i(s,a)$ extracts the ith feature from the state-action pair (s,a):

$$f(s,a) = \begin{pmatrix} f_1(s,a) \\ f_2(s,a) \\ \dots \\ f_n(s,a) \end{pmatrix}$$

   In the Freeway example, we have a vector with 14 state features times four actions. The function $f_i(s, Up)$ returns the closest car for row $i+1$ (for each of the 12 rows) if going up, $f_{13}(s, Up)$ is the distance to the goal for the first chicken, and $f_{14}(s, Up)$ the second chicken; e.g. $f_{13}(s,a) = \frac{row(s,chicken)}{max\_row}$.
2. A *weight* vector w of size $n \times |A|$: one weight for each feature-action pair. w_i^a defines the weight of a feature i for action a.

Defining State-Action Features

Often it is easier to just define features for states, rather than state-action pairs. The features are just a vector of n functions of the form $f_i(s)$.

However, for most applications, the weight of a feature is related to the action. The weight of being one step away from the end in Freeway is different if we go Up to if we go Right.

It is straightforward to construct $n \times |A|$ state-pair features from just n state features:

$$f_k(s,a) = \begin{cases} f_i(s) & \text{if } a = a_k \\ 0 & \text{otherwise} \quad 1 \le i \le n, 1 \le k \le |A| \end{cases}$$

This effectively results in $|A|$ different weight vectors: (s,a):

$$f(s,a_1) = \begin{pmatrix} f_{1,a_1}(s,a) \\ f_{2,a_1}(s,a) \\ 0 \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_2) = \begin{pmatrix} 0 \\ 0 \\ f_{1,a_2}(s,a) \\ f_{2,a_2}(s,a) \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_3) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ f_{1,a_3}(s,a) \\ f_{2,a_3}(s,a) \\ \dots \end{pmatrix} \quad \dots$$

Approximate Q-function Computation

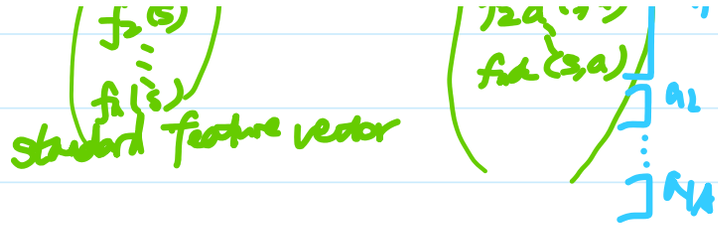
Give a feature vector f and a weight vector w, the Q-value of a state is a simple linear combination of features and weights:

$$\begin{aligned} Q(s,a) &= f_1(s,a) \cdot w_1^a + f_2(s,a) \cdot w_2^a + \dots + f_n(s,a) \cdot w_n^a \\ &= \sum_{i=0}^{n} f_i(s,a) w_i^a \end{aligned}$$

Example: For the Freeway example, we would assume that moving up would give a better score than moving down, all else equal (that is, if the closest car in the next row up is the same distance away than the closest in the next row down). So, for state s where the chicken is in row 1:

$$Q(s, Up) = f_1(s, Up) \cdot 0.31 + \dots + f_{14}(s, Up) \cdot 0.04$$

Note that to be effective, our feature values should be *normalised* using e.g. min-max normalisation or mean normalisation.



①Approximate Q(s,a) with linear regression

$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \dots + w_n f_n(s,a)$

Approximate Q-function Update

To use approximate Q-functions in reinforcement learning, there are two steps we need to change from the standard algorithms: (1) initialisation; and (2) update.

For initialisation, initialise all weights to 0. Alternatively, you can try Q-function initialisation and assign weights that you think will be 'good' weights.

For update, we now need to update the weights instead of the actions. For Q-learning, the update rule is now:

$$w_i \leftarrow w_i + \alpha[r + \gamma max'_a Q(s', a') - Q(s, a)]\, f_i(s, a)$$

For SARSA:

$$w_i^a \leftarrow w_i^a + \alpha[r + \gamma Q(s', a') - Q(s, a)]\, f_i(s, a)$$

Note: we need to update for each feature i for the last executed action a.

As this is linear, it will eventually converge!

Q-value Propagation

Note that this has the effect of updating Q-values to states that have never been visited!

In Freeway, for example, if we receive our first reward by crossing the road (going Up from the final row), this will update the weight all features for Up, and now we have a Q-value for going Up from *any* position on the final row.

Assume that all weights are 0, therefore, $Q(s, a) = 0$ for every state and action. Now, we receive the reward of 10 for getting to the other side of the road. If feature 14 is has the value $\frac{r}{D}$, where r is the current row and D is the distance to the other side, then we have:

$$\begin{aligned} w_i^a &\leftarrow w_i^a + \alpha[r + \gamma \max_a Q(s', a') - Q(s, a)]f_i(s, a) \\ w_{14}^{Up} &\leftarrow 0 + 0.5[10 + 0.9 \times 0]\tfrac{10}{10} \\ &= 5 \end{aligned}$$

From this, we now can get an estimate of $Q(s, Up)$ from any state because we have some weights in our linear function. Those that are closer to the other size of the road will get a higher Q-value than those further away (all other things being equal).

Q-function Approximation with Neural Networks: Deep Q-learning

The latest hype in reinforcement learning is all about the use of deep neural networks to approximate value and Q-functions.

In brief: instead of selecting features and training weights, we learn the parameters θ to a neural network. The Q-function is $Q(s, a; \theta)$, so takes the parameters as an argument.

The TD update for Q-learning is just:

$$\theta \leftarrow \theta + \alpha[r + \gamma \max_{a'} Q(s', a'; \theta) - Q(s, a; \theta)]\nabla_\theta Q(s, a; \theta)$$

Advantage (compared to linear Q functions): we do not need to select features – the 'features' will be learnt as part of the hidden layers.

Disadvantages: there are no convergence guarantees; and very data hungry because they need to learn features as well as Q-function.

Despite this, deep Q-learning often works remarkably well in practice, especially for tasks that require vision (see the robotic arm grasping unknown objects in the lecture on Q-learning).

Strengths and Limitations of Q-function Approximation

Approximating Q-functions using machine learning techniques has advantages and disadvantages.

Advantages:

- More efficient representation compared with Q-tables.
- Q-value propagation

Disadvantages:

- The Q-function is now only an approximation of the real Q-function: states that share feature values may have different actual values

linear vs deep
convergence
no feature selection
no convergence
data hungy

Applications of Deep Reinforcement Learning

A great application of using off-policy updates in deep Q-learning for robotic arms to learn how to grasp unknown objects. The only input for the problem is the camera data:

https://www.youtube.com/watch?v=cXaic_k80uM&feature=youtu.be

This is using policy iteration (policy gradient descent) rather than standard Q-learning.

What is reward shaping?

speed up training

In many applications, you will have some idea of what a good solution should look like. For example, in our simple navigation task, it is clear that moving towards the reward of +1 and away from the reward of -1 are likely to be good solutions.

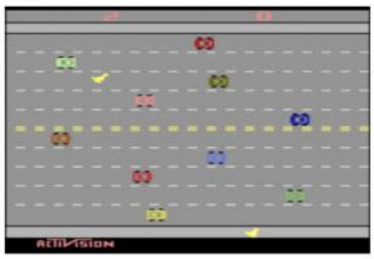
Can we then speed up learning and/or improve our final solution by nudging our reinforcement learner towards this behaviour?

The answer is: Yes! We can modify our reinforcement learning algorithm slightly to add domain knowledge, while also guaranteeing optimality.

This is known as *domain knowledge* — that is, stuff about the domain that the human modeller knows about while constructing the model to be solved.

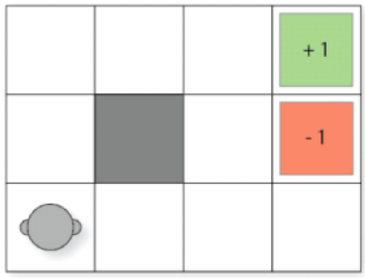
Exercise: Freeway

What would be a good reward for the Freeway game to learn how to get the chicken across the freeway?



Exercise: Gridworld

What would be a good reward for the GridWorld example?



Reward Shaping

In TD learning methods, we update a Q-function when a reward is received. E.g, for 1-step Q-learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

The approach to reward shaping is not to modify the reward function, but to just give additional reward for some moves:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \underbrace{F(s,s')}_{\text{additional reward}} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

The function F provides *heuristic* domain knowledge to the problem.

$r + F(s,s')$ is the *shaped reward* for an action.

$G^{\Phi} = \sum_{i=0}^{\infty} \gamma^i (r_i + F(s_i, s_{i+1}))$ is the shaped reward for the entire trace.

key idea: add some additional reward for particular behaviour

Potential-based Reward Shaping

Potential-based reward shaping is a particular type of reward shaping with nice theoretical guarantees. In potential-based reward shaping, F is of the form:

$$F(s,s') = \gamma\Phi(s') - \Phi(s)$$

We call Φ the *potential function* and $\Phi(s)$ is the potential of state s.

Theoretical guarantee: this will still converge to the optimal policy.

But! It may either increase or decrease the time taken to learn. A well-designed potential function will decrease the time.

Example: GridWorld

For Grid World, we can use 1 over Manhattan distance to define the potential function:

$$\Phi(s) = \frac{1}{|x(g) - x(s)| + |y(g) - y(s)|}$$

in which $x(s)$ and $y(s)$ return the x and y coordinates of the agent respectively, and g is the goal state.

Even on the very first iteration, a greedy policy, such as ϵ-greedy, will feedback those states closer to the +1 reward. From state (1,2) with $\gamma = 0.9$ if we go East, we get:

$$\begin{aligned} F((1,2),(2,2)) &= \gamma\Phi(2,2) - \Phi(1,2) \\ &= 0.9 \cdot \tfrac{1}{1} - \tfrac{1}{2} \\ &= 0.4 \end{aligned}$$

g=(2,2)

Reward Shaping Example: GridWorld (continued)

We can compare the Q-values for these states for the four different possible moves that could have been taken from (1,2), using and $\alpha = 0.5$ and $\gamma = 0.9$:

Action	r	$F(s,s')$	$\gamma \max_{a'} Q(s',a')$	New $Q(s,a)$
North	0	$0.9\frac{1}{2} - \frac{1}{2} = 0$	0	0
South	0	$0.9\frac{1}{2} - \frac{1}{2} = 0$	0	0
East	0	$0.9\frac{1}{1} - \frac{1}{2} = 0.4$	0	0.4
West	0	$0.9\frac{1}{3} - \frac{1}{2} = -0.2$	0	-0.1

Thus, we can see that our potential reward function rewards actions that go towards the goal and penalises actions that go away from the goal. Recall that state (1,2) is in the top row, so action North just leaves us in state (1,2) and South similarly because we cannot go into the wall.

But! It will not always work. Compare states (0,0) and (0,1). Our potential function will reward (0,1) because it is closer to the goal, but we know from the lecture on MDPs that (0,0) is a higher value state than (0,1). This is because our reward function does not consider the negative reward.

In practice, it is non-trivial to derive a perfect reward function. If we could, we would not need to even use reinforcement learning – we could just do a greedy search over the reward function.

Q-function initialisation

An approach related to reward shaping is *Q-function initialisation*. Recall that TD learning methods can start at any arbitrary Q-function. The closer our Q-function is to the optimal Q-function, the quicker it will converge.

Imagine if we happened to initialise our Q-function to the optimal Q-function. It would converge in one step!

Q-function initialisation is similar to reward shaping: we use heuristics to assign higher values to 'better' states. If we just define $\Phi(s) = V(s)$, then they are equivalent. In fact, if our potential function is *static* (the definition does not change during learning), then Q-function initialisation and reward shaping are equivalent[1].

[1] Wiewiora: ?Potential-based shaping and Q-value initialization are equivalent.? (JAIR, 2003)

Q-function Initialisation Example : GridWorld

Using the idea of inverse Manhattan distance, we can define an initial Q-function as follows for state (1,2):

$$\begin{aligned} Q((1,2), North) &= \tfrac{1}{5} - \tfrac{1}{5} &&= 0 \\ Q((1,2), South) &= \tfrac{1}{5} - \tfrac{1}{5} &&= 0 \\ Q((1,2), East) &= \tfrac{1}{1} - \tfrac{1}{2} &&= 0.5 \\ Q((1,2), West) &= \tfrac{1}{5} - \tfrac{1}{2} &&= -0.16^{*} \end{aligned}$$

Once we start learning over episodes, we will select those actions with a higher heuristic value, and also we are already closer to the optimal Q-function, so will will converge faster.

Summary

1. We can scale reinforcement learning by approximating Q-functions, rather than storing complete Q-tables.
2. Using simple linear methods in which we select features and learn weights are effective and guarantee convergence.
3. Using neural networks offer alternatives in which we do not need to select features, but require a lot of training data and have not convergence guarantees.

Some tips for reinforcement learning in the group project

1. Linear Q-function approximation should work well if the features are not strongly dependent on the random initial state.
2. I would be surprised if using deep Q networks for function approximation was effective because deep neural nets are data hungry and generating enough training data could require weeks or months of computation.
3. Design good reward shaping structures will help, but keep them simple.
4. For some advanced techniques, read Sutton and Barto; in particular stuff on: experience replay, approximate methods (even for value iteration!), and generalised policy iteration.

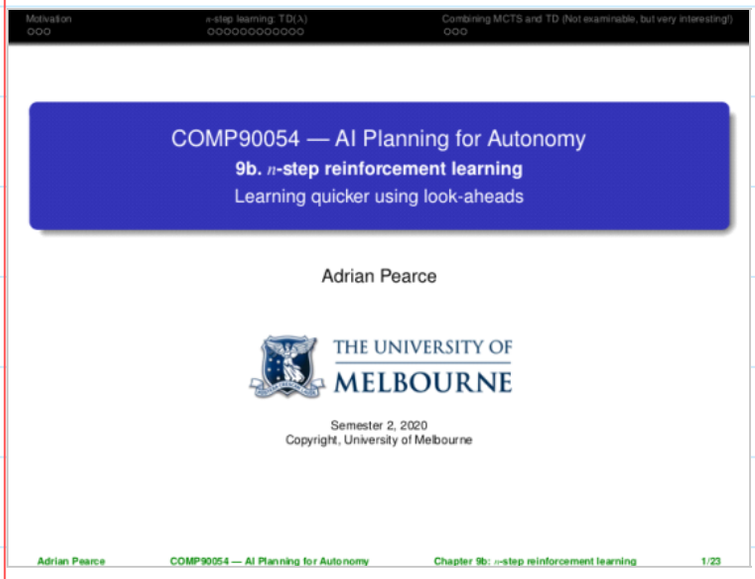
Reading

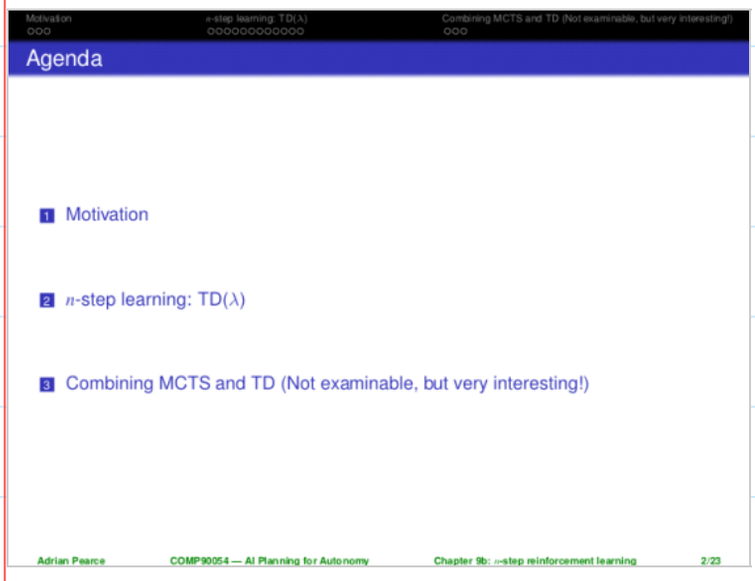
- (Chapter 9 On-policy Prediction with Approximation) of *Reinforcement Learning: An Introduction (Second Edition)* [*Sutton and Barto*, 2020]
  Available at:
  `http://www.incompleteideas.net/book/RLbook2020.pdf`
- *Playing Atari with Deep Reinforcement Learning* from DeepMind.
  Available at:
  `https://arxiv.org/pdf/1312.5602v1.pdf`
- Before AlphaGo there was TD-gammon, which was the first paper to combine reinforcement learning and neural networks:
  `http://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-003.pdf`

Chapter 12

2020年10月1日 星期四 14:48

intro09b-p
re-hando...





Learning Outcomes

1. Manually apply n-step reinforcement learning approximation to solve small-scale MDP problems given a set of
2. Design and implement n-step reinforcement learning to solve medium-scale MDP problems automatically
3. Argue the strengths and weaknesses of n-step reinforcement learning

Reinforcement Learning – Some Weaknesses

In the previous lecture, we looked at two fundamental temporal difference (TD) methods for reinforcement learning: Q-learning and SARSA.

These two methods have some weaknesses in this basic format:

1. Unlike Monte-Carlo methods, which reach a reward and then backpropagate this reward, TD methods use bootstrapping (they estimate the future discounted reward using $Q(s, a)$), which means that for problems with sparse rewards, it can take a long time to for rewards to propagate throughout a Q-function.
2. Both methods estimate a Q-function $Q(s, a)$, and the simplest way to model this is via a Q-table. However, this requires us to maintain a table of size $|A| \times |S|$, which is prohibitively large for any non-trivial problem.
3. Using a Q-table requires that we visit every reachable state many times and apply every action many times to get a good estimate of $Q(s, a)$. Thus, if we never visit a state s, we have no estimate of $Q(s, a)$, even if we have visited states that are very similar to s.
4. Rewards can be sparse, meaning that there are few state/actions that lead to non-zero rewards. This is problematic because initially, reinforcement learning algorithms behave entirely randomly and will struggle to find good rewards. Remember the Freeway demo from the previous lecture?

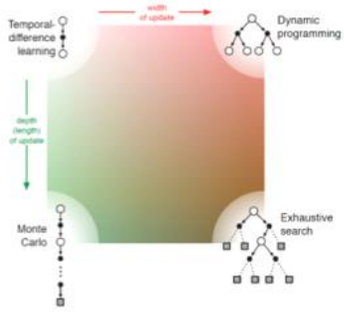
Reinforcement Learning – Some Improvements

To get around these limitations, we are going to look at three simple approaches that can improve temporal difference methods:

1. *n-step temporal difference learning*: Monte Carlo techniques execute entire traces and then backpropagate the reward, while basic TD methods only look at the reward in the next step, estimating the future wards. n-step methods instead look n steps ahead for the reward before updating the reward, and then estimate the remainder. *This lecture!*
2. *Approximate methods*: Instead of calculating an exact Q-function, we approximate it using simple methods that both eliminate the need for a large Q-table (therefore the methods scale better), and also allowing use to provide reasonable estimates of $Q(s, a)$ *even if we have not applied action a in state s previously. Last lecture!*
3. *Reward shaping and Value-Function Initialisation*: If rewards are sparse, we can modify/augment our reward function to reward behaviour that we think moves us closer to the solution, or we can guess the optimal Q-function and initial $Q(s, a)$ to be this. *Last lecture!*

Approaches to AI Planning and Reinforcement Learning

n-step TD learning comes from the idea used in the image below. Monte Carlo methods uses 'deep' updates and backups, where entire traces are executed and the reward backpropagated. Whereas temporal difference learning methods such as Q-learning and SARSA use 'shallow' updates and backups, only using the reward from the 1-step ahead. n-step learning finds the middle ground: only update the Q-function after having explored ahead n steps.



TD(λ)

We will look at TD(λ), in which λ is the parameter that determines n: the number of steps that we want to look ahead before updating the Q-function. Thus, TD(0) is just 'standard' reinforcement learning, and TD(1) looks one step beyond the immediate reward, TD(2) looks two steps beyond, etc.

Both Q-learning and SARSA have n-step version. We will look at TD(λ) more generally, and then show an algorithm for n-step SARSA. The version for Q-learning is similar.

Discounted Future Rewards (again)

When calculating a discounted reward over a trace, we can re-write as:

$$\begin{aligned} G_t &= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots \\ &= r_1 + \gamma(r_2 + \gamma(r_3 + \gamma(r_4 + \dots))) \end{aligned}$$

If G_t is the value received at time-step t, then $G_t = r_t + \gamma G_{t+1}$

In TD(0) methods such as Q-learning and SARSA, we do not know G_{t+1} when updating $Q(s,a)$, so we estimate using bootstrapping:

$$G_t = r_t + \gamma \cdot V(s_{t+1})$$

That is, the reward of the entire future from step t is estimated as the reward at t plus the estimated (discounted) future reward from $t+1$. $V(s_{t+1})$ is estimated using the maximum expected return (Q-learning) or the estimated value of the next action (SARSA).

This is a *one-step return*.

Truncated Discounted Rewards

However, we can estimate a two-step return:

$$G_t^2 = r_t + \gamma r_{t+1} + \gamma^3 V(s_{t+2})$$

or three-step return:

$$G_t^3 = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+2})$$

or n-step returns:

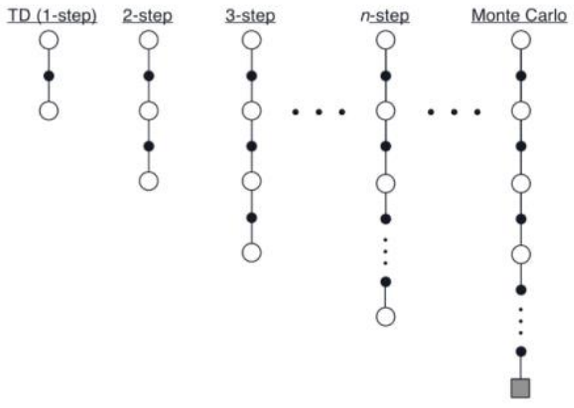
$$G_t^n = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \gamma^n V(s_{t+n})$$

In this above expression G_t^n is the full reward, *truncated* at n steps, at time t. The basic idea is that we do not update the Q-value immediately after executing an action: we wait n steps and update it based on the n-step return.

If T is the termination step and $t + n > T$, then we just use the full reward.

In Monte-Carlo methods, we go all the way to the end of an episode. Monte-Carlo Tree Search is one such Monte-Carlo method, but there are others that we do not cover.

Different Levels Truncated Rewards



Updating the Q-function

The update rule for the Q-function is then different. First, we need to calculate the truncated reward for n steps, in which τ is the time step that we are updating for:

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i$$

This just sums the rewards from time step $\tau + 1$ until either n steps $(\tau + n)$ or termination of the episode (T), whichever comes first. For n-step SARSA, we have:

Then update:

If $\tau + n < T$ then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$
$Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$

The first line just adds the future expect reward if we are not at the end of the episode (if $\tau + n < T$).

$Q(s,a) \leftarrow Q(s,a) + \alpha[G_t + \gamma Q(s',a')]$

SARSA

But! It's not so simple

While conceptually this is not so difficult, we have to modify our algorithms quite a bit, because at each step, n-step return uses a reward from the future.

For the first $n-1$ steps of the any episode, we do not update Q at all.

Also, we have to continue updating $n-1$ steps after the end of the episode.

This leads to the algorithm on the following slide. All of the changes, except for the three lines immediately after 'if $\tau \geq 0$', just manage the n-steps.

Computationally, this is not much worse than 1-step learning. We need to store the last n states, but the per-step computation is small and uniform for n-step, just as for 1-step.

n-step SARSA

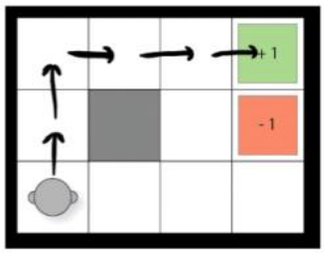
n-step Sarsa for estimating $Q \approx q_*$, or $Q \approx q_\pi$ for a given π

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$
Initialize π to be ε-greedy with respect to Q, or to a fixed given policy
Parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$, a positive integer n
All store and access operations (for S_t, A_t, and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
    If $t < T$, then:
      Take action A_t
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t+1$
      else:
        Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$
    $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ $(G_{\tau:\tau+n})$
      $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha\,[G - Q(S_\tau, A_\tau)]$
      If π is being learned, then ensure that $\pi(\cdot|S_\tau)$ is ε-greedy wrt Q
  Until $\tau = T - 1$

Exercise

Consider our simple 2D navigation task, in which we do not know the probability transitions nor the rewards. Initially, the reinforcement learning algorithm will be required to search randomly until it finds a reward. Propagated this reward back n-steps will be helpful.



Assuming $Q(s,a) = 0$ for all s and a, if we (finally) traverse the episode the labelled episode, what will our Q-function look like for a 5-step update with $\alpha = 0.5$ and $\gamma = 0.9$?

Exercise (continued)

We only receive a reward in the last action, and all other actions give an immediate reward of 0 until then:

$$
\begin{aligned}
G &\leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_i \\
G_1 &\leftarrow \gamma^1 \cdot 0 + \ldots + \gamma^5 \cdot 1 \\
&\leftarrow 0.9^5 \cdot 1 \\
&\leftarrow 0.5905
\end{aligned}
$$

So, we update the Q-value for the state $(0, 2)$, which is 5 steps back:

$$
\begin{aligned}
Q((0,2),E) &\leftarrow Q((0,2),E) + \alpha[G_1 - Q((0,2),E)] \\
&\leftarrow 0 + 0.5[0.5905 - 0] \\
&\leftarrow 0.2953
\end{aligned}
$$

SARSA

$Q \leftarrow 0 + 0.5(0.5905 + 0 - 0)$

discount future reward

Exercise (continued)

The table below compares 1-step vs. 5-step SARSA for the trace above. In 1-step SARSA, reaching the reward only informs the state from which it is reached. Whereas for 5-step, it informs the previous five steps. Then, in the next episode, there is more chance of encountering a non-zero state, so which will again inform the five steps instead of just one. The rewards 'spread' throughout the Q-table faster.

With 1-step learning

State	Action			
	North	South	East	West
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0	0
...				
(1,2)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0.45	0
(2,3)	0	0	0	0
...				

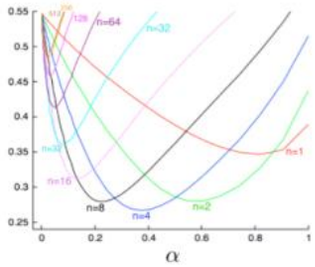
With 5-step learning

State	Action			
	North	South	East	West
(0,0)	0	0	0	0
(0,1)	0	0	0	0
(0,2)	0	0	0.2953	0
...				
(1,2)	0	0	0.3281	0
(2,1)	0.405	0	0	0
(2,2)	0	0.3645	0.45	0
(2,3)	0	0	0	0
...				

Simple experiment: Random Walk

Consider the following simple deterministic Markov reward process:

The following shows results from a series of experiments in varying α and n. The y-axis shows the root mean-squared error:



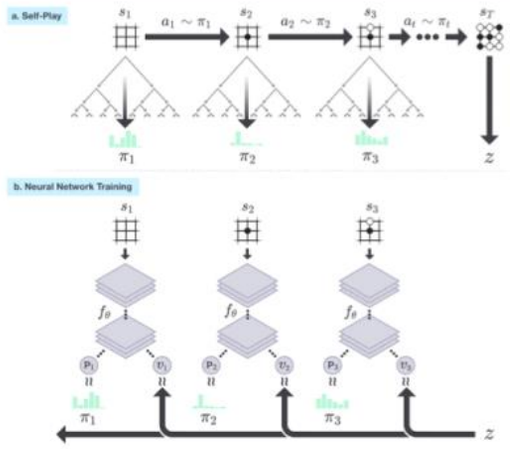
$n = 1$ is TD(0), while larger n are closer to Monte-Carlo methods. Note that the 'in between' parameters perform best in this example.

Combining MCTS and Reinforcement Learning: AlphaGo Zero (Not examinable)

AlphaGo Zero (or more accurately its predecessor AlphaGo) made headlines when it beat Go world champion Lee Sodol in 2016. It uses a combination of MCTS and (deep) reinforcement learning to learn a policy. A simple overview:

1. It uses a deep neural network to estimate the Q-function. More accurately, it gives an estimate of the probability of selecting action a in state s ($P(a|s)$), and the *value* of the state ($V(s)$), which represents the probability of the player winning from s.
2. It is trained via *self-play*.
3. At each move, AlphaGo Zero:
   1. Executes an MCTS search using UCB $Q(s,a) + P(s,a)/1 + N(s,a)$, which returns the probabilities of playing each move.
   2. The neural network is used to guide the MCTS by influencing $Q(s,a)$.
   3. The final result of a simulated game is used as the reward for each simulation.
   4. After a set number of MCTS simulations, the best move is chosen for self-play.
   5. Repeat steps 1-4 for each move until the self-play game ends.
   6. Then, feedback the result of the self-play game to update the Q function for each move.

AlphaGo is Best Summarised Using This Figure



Reading

- (Chapter 7 n-step Bootstrapping & Chapter 12 Eligibility Traces) of *Reinforcement Learning: An Introduction (Second Edition)* [*Sutton and Barto, 2020*]
  Available at:
  `http://www.incompleteideas.net/book/RLbook2020.pdf`
- *Mastering the Game of Go without Human Knowledge* from DeepMind.
  Available at:
  `https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge` *(select Download link for pdf)*

Chapter 13

2020年10月15日 星期四 16:18

intro11-pre-handout

A Brief Introduction to the Basics of Game Theory

Matthew O. Jackson, Stanford University

I provide a (very) brief introduction to game theory. I have developed these notes to provide quick access to some of the basics of game theory; mainly as an aid for students in courses in which I assumed familiarity with game theory but did not require it as a prerequisite. Of course, the material discussed here is only the proverbial tip of the iceberg, and there are many sources that offer much more complete treatments of the subject.[1] Here, I only cover a few of the most fundamental concepts, and provide just enough discussion to get the ideas across without discussing many issues associated with the concepts and approaches. Fuller coverage is available through a free on-line course that can be found via my website: http://www.stanford.edu/~jacksonm/

The basic elements of performing a noncooperative [2] game-theoretic analysis are (1) framing the situation in terms of the actions available to players and their payoffs as a function of actions, and (2) using various equilibrium notions to make either descriptive or

[1] For graduate-level treatments, see Roger Myerson's (1991) Game Theory: Analysis of Conflict, Cambridge, Mass.: Harvard University Press; Ken Binmore's (1992) Fun and Games, Lexington, Mass.: D.C. Heath; Drew Fudenberg and Jean Tirole's (1993) Game Theory, Cambridge, Mass.: MIT Press; and Martin Osborne and Ariel Rubinstein's (1994) A Course in Game Theory, Cambridge, Mass.: MIT Press. There are also abbreviated texts offering a quick tour of game theory, such as Kevin Leyton-Brown and Yoav Shoham's (2008) Essentials of Game Theory, Morgan and Claypool Publishers. For broader readings and undergraduate level texts, see R. Duncan Luce and Howard Raiffa (1959) Games and Decisions: Introduction and Critical Survey; Robert Gibbons (1992) Game Theory for Applied Economists; Colin F. Camerer (2003) Behavioral Game Theory: Experiments in Strategic Interaction; Martin J. Osborne (2003) An Introduction to Game Theory; Joel Watson (2007) Strategy: An Introduction to Game Theory; Avinash K. Dixit and Barry J. Nalebuff (2010) The Art of Strategy: A Game Theorist's Guide to Success in Business and Life; Joseph E. Harrington, Jr. (2010) Games, Strategies, and Decision Making, Worth Publishing.

[2] "Noncooperative game theory" refers to models in which each players are assumed to behave selfishly and their behaviors are directly modeled. "Cooperative game theory," which I do not cover here, generally refers to more abstract and axiomatic analyses of bargains or behaviors that players might reach, without explicitly modeling the processes. The name "cooperative" derives in part from the fact that the analyses often (but not always) incorporate coalitional considerations, with important early analyses appearing in John von Neumann and Oskar Morgenstern's 1944 foundational book "Theory of Games and Economic Behavior."

prescriptive predictions. In framing the analysis, a number of questions become important. First, who are the players? They may be people, firms, organizations, governments, ethnic groups, and so on. Second, what actions are available to them? All actions that the players might take that could affect any player's payoffs should be listed. Third, what is the timing of the interactions? Are actions taken simultaneously or sequentially? Are interactions repeated? The order of play is also important. Moving after another player may give player i an advantage of knowing what the other player has done; it may also put player i at a disadvantage in terms of lost time or the ability to take some action. What information do different players have when they take actions? Fourth, what are the payoffs to the various players as a result of the interaction? Ascertaining payoffs involves estimating the costs and benefits of each potential set of choices by all players. In many situations it may be easier to estimate payoffs for some players (such as yourself) than others, and it may be unclear whether other players are also thinking strategically. This consideration suggests that careful attention be paid to a sensitivity analysis.

Once we have framed the situation, we can look from different players' perspectives to analyze which actions are optimal for them. There are various criteria we can use.

1 Games in Normal Form

Let us begin with a standard representation of a game, which is known as a *normal form* game, or a game in *strategic form*:

- The set of players is $N = \{1, \ldots, n\}$.
- Player i has a set of actions, a_i, available. These are generally referred to as *pure strategies*.[3] This set might be finite or infinite.
- Let $a = a_1 \times \cdots \times a_n$ be the set of all *profiles* of pure strategies or actions, with a generic element denoted by $a = (a_1, \ldots, a_n)$.

[3] The term "pure" indicates that a single action is chosen, in contrast with "mixed" strategies that I discuss below, in which there is a randomization over actions.

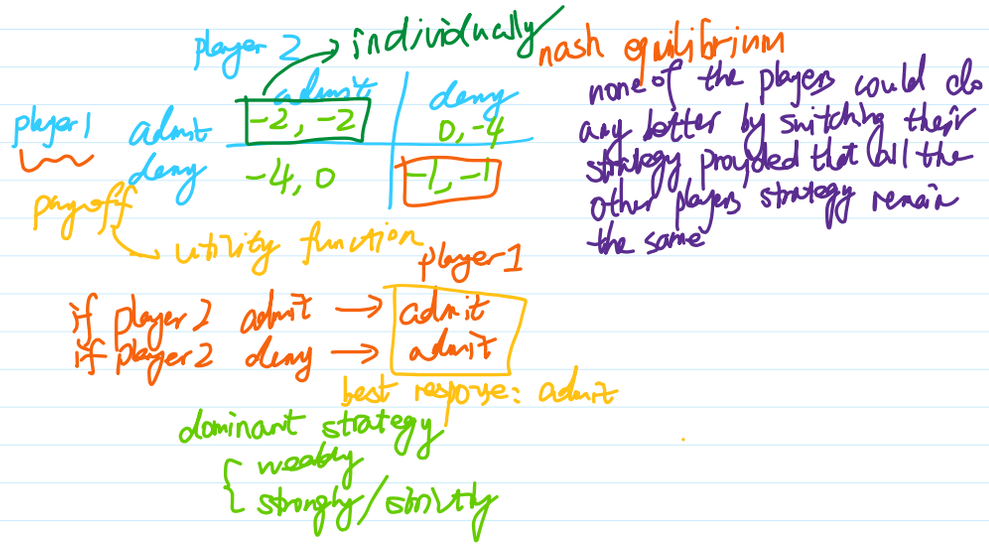
- Player i's payoff as a function of the vector of actions taken is described by a function $u_i : A \to \mathbb{R}$, where $u_i(a)$ is i's payoff if the a is the profile of actions chosen in the society.

Normal form games are often represented by a table. Perhaps the most famous such game is the *prisoners' dilemma*, which is represented in Table 1. In this game there are two players who each have two pure strategies, where $a_i = \{C, D\}$, and C stands for "cooperate" and D stands for "defect." The first entry indicates the payoff to the row player (or player 1) as a function of the pair of actions, while the second entry is the payoff to the column player (or player 2).

Table 1: A Prisoners' Dilemma Game

		Player 2	
		C	D
Player 1	C	-1, -1	-3, 0
	D	0, -3	-2, -2

The usual story behind the payoffs in the prisoners' dilemma is as follows. The two players have committed a crime and are now in separate rooms in a police station. The prosecutor has come to each of them and told them each: "If you confess and agree to testify against the other player, and the other player does not confess, then I will let you go. If you both confess, then I will send you both to prison for 2 years. If you do not confess and the other player does, then you will be convicted and I will seek the maximum prison sentence of 3 years. If nobody confesses, then I will charge you with a lighter crime for which we have



- Player i's payoff as a function of the vector of actions taken is described by a function $u_i : A \to \mathbb{R}$, where $u_i(a)$ is i's payoff if the a is the profile of actions chosen in the society.

Normal form games are often represented by a table. Perhaps the most famous such game is the *prisoners' dilemma*, which is represented in Table 1. In this game there are two players who each have two pure strategies, where $a_i = \{C, D\}$, and C stands for "cooperate" and D stands for "defect." The first entry indicates the payoff to the row player (or player 1) as a function of the pair of actions, while the second entry is the payoff to the column player (or player 2).

Table 1: A Prisoners' Dilemma Game

		Player 2	
		C	D
Player 1	C	-1, -1	-3, 0
	D	0, -3	-2, -2

The usual story behind the payoffs in the prisoners' dilemma is as follows. The two players have committed a crime and are now in separate rooms in a police station. The prosecutor has come to each of them and told them each: "If you confess and agree to testify against the other player, and the other player does not confess, then I will let you go. If you both confess, then I will send you both to prison for 2 years. If you do not confess and the other player does, then you will be convicted and I will seek the maximum prison sentence of 3 years. If nobody confesses, then I will charge you with a lighter crime for which we have enough evidence to convict you and you will each go to prison for 1 year." So the payoffs in the matrix represent time lost in terms of years in prison. The term *cooperate* refers to cooperating with the other player. The term *defect* refers to confessing and agreeing to testify, and so breaking the (implicit) agreement with the other player.

Note that we could also multiply each payoff by a scalar and add a constant, which is an equivalent representation (as long as all of a given player's payoffs are rescaled in the same

way). For instance, in Table 2 I have doubled each entry and added 6. This transformation leaves the strategic aspect of the game unchanged.

Table 2: A Rescaling of the Prisoners' Dilemma

		Player 2	
		C	D
Player 1	C	4, 4	0, 6
	D	6, 0	2, 2

There are many games that might have different descriptions motivating them but have a similar normal form in terms of the strategic aspects of the game. Another example of the same game as the prisoners' dilemma is what is known as a *Cournot duopoly*. The story is as follows. Two firms produce identical goods. They each have two production levels, high or low. If they produce at high production, they will have a lot of the goods to sell, while at low production they have less to sell. If they cooperate, then they agree to each produce at low production. In this case, the product is rare and fetches a very high price on the market, and they each make a profit of 4. If they each produce at high production (or defect), then they will depress the price, and even though they sell more of the goods, the price drops sufficiently to lower their overall profits to 2 each. If one defects and the other cooperates, then the price is in a middle range. The firm with the higher production sells more goods and earns a higher profit of 6, while the firm with the lower production just covers its costs and earns a profit of 0.

1.1 Dominant Strategies

Given a game in normal form, we then can make predictions about which actions will be chosen. Predictions are particularly easy when there are "dominant strategies." A dominant strategy for a player is one that produces the highest payoff of any strategy available *for every possible action by the other players*.

That is, a strategy $a_i \in a_i$ is a *dominant* (or weakly dominant) strategy for player i if

$u_i(a_i, a_{-i}) \geq u_i(a'_i, a_{-i})$ for all a'_i and all $a_{-i} \in a_{-i}$. A strategy is a *strictly dominant strategy* if the above inequality holds strictly for all $a'_i \neq a_i$ and all $a_{-i} \in a_{-i}$.

Dominant strategies are powerful from both an analytical point of view and a player's perspective. An individual does not have to make any predictions about what other players might do, and still has a well-defined best strategy.

In the prisoners' dilemma, it is easy to check that each player has a strictly dominant strategy to defect—that is, to confess to the police and agree to testify. So, if we use dominant strategies to predict play, then the unique prediction is that each player will defect, and both players fare worse than for the alternative strategies in which neither defects. A basic lesson from the prisoners' dilemma is that individual incentives and overall welfare need not coincide. The players both end up going to jail for 2 years, even though they would have gone to jail for only 1 year if neither had defected. The problem is that they cannot trust each other to cooperate: no matter what the other player does, a player is best off defecting.

Note that this analysis presumes that all relevant payoff information is included in the payoff function. If, for instance, a player fears retribution for confessing and testifying, then that should be included in the payoffs and can change the incentives in the game. If the player cares about how many years the other player spends in jail, then that can be written into the payoff function as well.

When dominant strategies exist, they make the game-theoretic analysis relatively easy. However, such strategies do not always exist, and then we can turn to notions of equilibrium.

1.2 Nash Equilibrium

A pure strategy Nash equilibrium[4] is a profile of strategies such that each player's strategy is a best response (results in the highest available payoff) against the equilibrium strategies of the other players.

[4]The concept is named after John Nash, who provided the first existence proof in finite games: Nash, J.F. (1951) Non-Cooperative Games, Annals of Mathematics 54:286293. On occasion it is also referred to as *Cournot–Nash equilibrium*, with reference to Antoine Augustin Cournot, who in the 1830's first developed such an equilibrium concept in the analysis of oligopoly (a set of firms in competition with one another) – Cournot (1838) Recherches sur les principes mathematiques de la theorie des richesses, translated as Researches into the Mathematical Principles of the Theory of Wealth, New York: Macmillan (1897).

A strategy a_i is a *best reply*, also known as a *best response*, of player i to a profile of strategies $a_{-i} \in a_{-i}$, for the other players if

$$u_i(a_i, a_{-i}) \geq u_i(a'_i, a_{-i})$$

for all a'_i. A best response of player i to a profile of strategies of the other players is said to be a *strict best response* if it is the unique best response.

A profile of strategies $a \in A$ is a *pure strategy Nash equilibrium* if a_i is a best reply to a_{-i} for each i. That is, a is a Nash equilibrium if

$$u_i(a_i, a_{-i}) \geq u_i(a'_i, a_{-i})$$

for all i and a'_i. This definition might seem somewhat similar to that of dominant strategy, but there is a critical difference. A pure strategy Nash equilibrium only requires that the action taken by each agent be best against the actual equilibrium actions taken by the other players, and not necessarily against all possible actions of the other players.

A Nash equilibrium has the nice property that it is stable: if each player expects a to be the profile of actions played, then no player has any incentive to change his or her action. In other words, no player regrets having played the action that he or she played in a Nash equilibrium.

In some cases, the best response of a player to the actions of others is unique. A Nash equilibrium such that all players are playing actions that are unique best responses is called a *strict* Nash equilibrium. A profile of dominant strategies is a Nash equilibrium but not vice versa.

To see another illustration of Nash equilibrium, consider the following game between two firms that are deciding whether to advertise. Total available profits are 28, to be split between the two firms. Advertising costs a firm 8. Firm 1 currently has a larger market share than firm 2, so it is seeing 16 in profits while firm 2 is seeing 12 in profits. If they both advertise, then they will split the market evenly and get 14 in base profits each, but then must also pay the costs of advertising, so they receive see net profits of 6 each. If one advertises while the other does not, then the advertiser captures three-quarters of the market (but also pays for advertising) and the non-advertiser gets one-quarter of the market. (There

are obvious simplifications here: just considering two levels of advertising and assuming that advertising only affects the split and not the total profitability.) The net payoffs are given in the Table 3.

Table 3: An Advertising Game

		Firm 2	
		Not	Ads
Firm 1	Not	16, 12	7, 13
	Ads	13, 7	6, 6

To find the equilibrium, we have to look for a pair of actions such that neither firm wants to change its action given what the other firm has chosen. The search is made easier in this case, since firm 1 has a strictly dominant strategy of not advertising. Firm 2 does not have a dominant strategy; which strategy is optimal for it depends on what firm 1 does. But given the prediction that firm 1 will not advertise, firm 2 is best off advertising. This forms a Nash equilibrium, since neither firm wishes to change strategies. You can easily check that no other pairs of strategies form an equilibrium.

While each of the previous games provides a unique prediction, there are games in which there are multiple equilibria. Here are three examples.

EXAMPLE 1 *A Stag Hunt Game The first is an example of a coordination game, as depicted in Table 4. This game might be thought of as selecting between two technologies, or coordinating on a meeting location. Players earn higher payoffs when they choose the same action than when they choose different actions. There are two (pure strategy) Nash equilibria: (S, S) and (H, H).*

This game is also a variation on Rousseau's "stag hunt" game.[5] The story is that two hunters are out, and they can either hunt for a stag (strategy S) or look for hares (strategy H). Succeeding in getting a stag takes the effort of both hunters, and the hunters are separated

[5]To be completely consistent with Rousseau's story, (H, H) should result in payoffs of (3, 3), as the payoff to hunting for hare is independent of the actions of the other player in Rousseau's story.

Table 4: A Coordination Game

		Player 2	
		S	H
Player 1	S	5, 5	0, 3
	H	3, 0	4, 4

in the forest and cannot be sure of each other's behavior. If both hunters are convinced that the other will hunt for stag, then hunting stag is a strict or unique best reply for each player. However, if one turns out to be mistaken and the other hunter hunts for hare, then one will go hungry. Both hunting for hare is also an equilibrium and hunting for hare is a strict best reply if the other player is hunting for hare. This example hints at the subtleties of making predictions in games with multiple equilibria. On the one hand, (S, S) (hunting stag by both) is a more attractive equilibrium and results in high payoffs for both players. Indeed, if the players can communicate and be sure that the other player will follow through with an action, then playing (S, S) is a stable and reasonable prediction. However, (H, H) (hunting hare by both) has properties that make it a useful prediction as well. It does not offer as high a payoff, but it has less risk associated with it. Here playing H guarantees a minimum payoff of 3, while the minimum payoff to S is 0. There is an extensive literature on this subject, and more generally on how to make predictions when there are multiple equilibria.[6]

EXAMPLE 2 *A "Battle of the Sexes" Game The next example is another form of coordination game, but with some asymmetries in it. It is generally referred to as a "battle of the sexes" game, as depicted in Table 5.*

The players have an incentive to choose the same action, but they each have a different favorite action. There are again two (pure strategy) Nash equilibria: (X, X) and (Y, Y). Here, however, player 1 would prefer that they play equilibrium (X, X) and player 2 would prefer (Y, Y). The battle of the sexes title refers to a couple trying to coordinate on where to meet for a night out. They prefer to be together, but also have different preferred outings.

[6]See, for example, the texts cited in Footnote 1.

mixed
Eq> 1L=P
Eq> (R)=3(1-P)
P=3(1-P)
⇒P=3/4

Table 5: A "battle of the sexes" game

		Player 2 X	Player 2 Y
Player 1	X	3, 1	0, 0
	Y	0, 0	1, 3

P
1-P

EXAMPLE 3 *Hawk-Dove and Chicken Games There are also what are known as anti-coordination games, with the prototypical version being what is known as the hawk-dove game or the chicken game, with payoffs as in Table 6.*

Table 6: A "hawk-dove" game

		Player 2 Hawk	Player 2 Dove
Player 1	Hawk	0, 0	3, 1
	Dove	1, 3	2, 2

Here there are two pure strategy equilibria, (Hawk, Dove) and (Dove, Hawk). Players are in a potential conflict and can be either aggressive like a hawk or timid like a dove. If they both act like hawks, then the outcome is destructive and costly for both players with payoffs of 0 for both. If they each act like doves, then the outcome is peaceful and each gets a payoff of 2. However, if the other player acts like a dove, then a player would prefer to act like a hawk and take advantage of the other player, receiving a payoff of 3. If the other player is playing a hawk strategy, then it is best to play a dove strategy and at least survive rather than to be hawkish and end in mutual destruction.

1.3 Randomization and Mixed Strategies

In each of the above games, there was at least one pure strategy Nash equilibrium. There are also simple games for which pure strategy equilibrium do not exist. To see this, consider the

Assumptions
① players are rational
② Everyone is self-interest
③ Perfect information
④ Pay-offs are known
⑤ Simultaneous moves

following simple variation on a penalty kick in a soccer match. There are two players: the player kicking the ball and the goalie. Suppose, to simplify the exposition, that we restrict the actions to just two for each player (there are still no pure strategy equilibria in the larger game, but this simplified version makes the exposition easier). The kicking player can kick to the left side or to the right side of the goal. The goalie can move to the left side or to the right side of the goal and has to choose before seeing the kick, as otherwise there is too little time to react. To keep things simple, assume that if the player kicks to one side, then she scores for sure if the goalie goes to the other side, while the goalie is certain to save it if the goalie goes to the same side. The basic payoff structure is depicted in Table 7.

Table 7: A Penalty-Kick Game.

		Goalie L	Goalie R
Kicker	L	-1, 1	1, -1
	R	1, -1	-1, 1

P
1-P

no nash eq.

This is also the game known as "matching pennies." The goalie would like to choose a strategy that matches that of the kicker, and the kicker wants to choose a strategy that mismatches the goalie's strategy.[7]

It is easy to check that no pair of pure strategies forms an equilibrium. What is the solution here? It is just what you see in practice: the kicker randomly picks left versus right, in this particular case with equal probability, and the goalie does the same. To formalize this observation we need to define randomized strategies, or what are called *mixed strategies*. For ease of exposition suppose that a_i is finite; the definition extends to infinite strategy spaces with proper definitions of probability measures over pure actions.

[7]For an interesting empirical test of whether goalies and kickers on professional soccer teams randomize properly, see Chiappori, Levitt, and Groseclose (2002) Testing Mixed-Strategy Equilibria When Players Are Heterogeneous: The Case of Penalty Kicks in Soccer, American Economic Review 92(4):1138 - 1151; and see Walker and Wooders (2001) Minimax Play at Wimbledon, American Economic Review 91(5):1521 - 1538, for an analysis of randomization in the location of tennis serves in professional tennis matches.

A mixed strategy for a player i is a distribution s_i on a_i, where $s_i(a_i)$ is the probability that a_i is chosen. A profile of mixed strategies $(s_1, \ldots, s_n)$ forms *a mixed-strategy Nash equilibrium* if

$$\sum_{a} \left(\prod_j s_j(a_j) \right) u_i(a) \geq \sum_{a} \left(\prod_{j \neq i} s_j(a_j) \right) u_i(a'_i, a_{-i})$$

for all i and a'_i.

So a profile of mixed strategies is an equilibrium if no player has some strategy that would offer a better payoff than his or her mixed strategy in reply to the mixed strategies of the other players. Note that this reasoning implies that a player must be indifferent to each strategy that he or she chooses with a positive probability under his or her mixed strategy. Also, players' randomizations are independent.[8] A special case of a mixed strategy is a pure strategy, where probability 1 is placed on some action.

It is easy to check that each mixing with probability 1/2 on L and R is the unique mixed strategy of the matching pennies game above. If a player, say the goalie, places weight of more than 1/2 on L, for instance, then the kicker would have a best response of choosing R with probability 1, but then that could not be an equilibrium as the goalie would want to change his or her action, and so forth.

There is a deep and long-standing debate about how to interpret mixed strategies, and the extent to which people really randomize. Note that in the goalie and kicker game, what is important is that each player not know what the other player will do. For instance, it could be that the kicker decided before the game that if there was a penalty kick then she would kick to the left. What is important is that the kicker not be known to always kick to the left.[9]

We can begin to see how the equilibrium changes as we change the payoff structure. For example, suppose that the kicker is more skilled at kicking to the right side than to the left.

[8]An alternative definition of correlated equilibrium allows players to use correlated strategies but requires some correlating device that only reveals to each player his or her prescribed strategy and that these are best responses given the conditional distribution over other players' strategies.

[9]The contest between pitchers and batters in baseball is quite similar. Pitchers make choices about the location, velocity, and type of pitch (e.g., whether various types of spin are put on the ball). If a batter knows what pitch to expect in a given circumstance, that can be a significant advantage. Teams scout one another's players and note any tendencies or biases that they might have and then try to respond accordingly.

In particular, keep the payoffs as before, but now suppose that the kicker has an even chance of scoring when kicking right when the goalie goes right. This leads to the payoffs in Table 8.

Table 8: A biased penalty-kick game

		Goalie L	Goalie R
Kicker	L	-1, 1	1, -1
	R	1, -1	0, 0

What does the equilibrium look like? To calculate the equilibrium, it is enough to find a strategy for the goalie that makes the kicker indifferent, and a strategy for the kicker that makes the goalie indifferent.[10]

Let s_1 be the kicker's mixed strategy and s_2 be the goalie's mixed strategy. It must be that the kicker is indifferent. The kicker's expected payoff from kicking L is $-1 \cdot s_2(L) + 1 \cdot s_2(R)$ [11] and the payoff from R is $1 \cdot s_2(L) + 0 \cdot s_2(R)$, so that indifference requires that

$$-s_2(L) + s_2(R) = s_2(L),$$

which implies that $2s_2(L) = s_2(R)$. Since these must sum to one (as they are probabilities), this implies that $s_2(L) = 1/3$ and $s_2(R) = 2/3$. Similar calculations based on the requirement that the goalie be indifferent lead to

$$s_1(L) - s_1(R) = -s_1(L),$$

[10]This reasoning is a bit subtle, as we are not directly choosing actions that maximize the goalie's payoff and maximize the kicker's payoff, but instead are looking for a mixture by one player that makes the other indifferent. This feature of mixed strategies takes a while to grasp, but experienced players seem to understand it well, as discussed below.

[11]To see where this payoff comes from, note that there is a $s_2(L)$ chance that the goalie also goes L and then the kicker loses and gets a payoff of -1, and a $s_2(R)$ chance that the goalie goes right and then the kicker wins and gets a payoff of 1; thus the expected payoff is $-1 \cdot s_2(L) + 1 \cdot s_2(R)$

and so the kicker's equilibrium strategy must satisfy $2s_1(L) = s_1(R)$, which this implies that $s_1(L) = 1/3$ and $s_1(R) = 2/3$.

Note that as the kicker gets more skilled at kicking to the right, they both adjust to using the right strategy more. The goalie ends up using the R strategy with higher probability than before even though that strategy has gotten worse for the goalie in terms of just looking at each entry of Table 8 compared to Table 7. This reflects the strategic aspect of the game: each player's strategy reacts to the other's strategy, and not just absolute changes in payoffs as one might superficially expect. The kicker using R more means that the goalie is still indifferent with the new payoffs, and the goalie has to adjust to using R more in order to keep the kicker indifferent.[12]

While not all games have pure strategy Nash equilibrium, every game with a finite set of actions has at least one mixed strategy Nash equilibrium (with a special case of a mixed strategy equilibrium being a pure strategy equilibrium), as shown in the seminal paper by John Nash (1951) Non-Cooperative Games, Annals of Mathematics 54:286 - 295.

2 Sequentiality, Extensive Form Games, and Backward Induction

Let us now turn to the question of timing. In the above discussion it was implicit that each player was selecting a strategy with beliefs about the other players' strategies but without knowing exactly what they were.

If we wish to be more explicit about timing, then we can consider what are known as games in *extensive form*, which include a complete description of who moves in what order and what they have observed when they move.[13] There are advantages to working with

[12]Interestingly, there is evidence that professional soccer players are better at playing games that have mixed strategy equilibria than are people with less experience in such games, which is consistent with this observation (see Palacios-Huerta and Volij (2008) "Experientia Docet: Professionals Play Minimax in Laboratory Experiments," *Econometrica*, 76:1, pp 71 - 115.

[13]One can collapse certain types of extensive form games into normal form by simply defining an action to be a complete specification of how an agent would act in all possible contingencies. Agents then choose these actions simultaneously at the beginning of the game. But the normal form becomes more complicated

extensive form games, as they allow more explicit treatments of timing and for equilibrium concepts that require credibility of strategies in response to the strategies of others.

Definitions for a general class of extensive form games are notationally intensive. Here I will just discuss a special class of extensive form games—finite games of perfect information—which allows for a treatment that avoids much of the notation. These are games in which players move sequentially in some pre-specified order (sometimes contingent on which actions have been chosen), each player moves at most a finite number of times, and each player is completely aware of all moves that have been made previously. These games are particularly well behaved and can be represented by simple trees, where a "nonterminal" node is associated with the move of a specified player and an edge corresponds to different actions the player might take, and "terminal" nodes (that have no edges following them) list the payoffs if those nodes are reached, as in Figure 1. I will not provide formal definitions, but simply refer directly to games representable by such finite game trees.

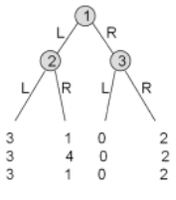


Figure 1: A Game Tree with 3 Players and Two Actions Each.

Each node has a player's label attached to it. There is an identified *root node* that corresponds to the first player to move (player 1 in Figure 1) and then subsequent nodes

than the two-by-two games discussed above.

that correspond to subsequent players who make choices. In Figure 1, player 1 has a choice of moving either left or right. The branches in the tree correspond to the different actions available to the player at a given node. In this game, if player 1 moves left, then player 2 moves next; while if player 1 moves right, then player 3 moves next. It is also possible to have trees in which player 1 chooses twice in a row, or no matter what choice a given player makes it is a certain player who follows, and so forth. The payoffs are given at the end nodes and are listed for the respective players. The top payoff is for player 1, the second for player 2, and the bottom for player 3. So the payoffs depend on the set of actions taken, which then determines a path through the tree. An equilibrium provides a prediction about how each player will move in each contingency and thus makes a prediction about which path will be taken; we refer to that prediction as the *equilibrium path*.

We can apply the concept of a Nash equilibrium to such games, which here is a specification of what each player would do at each node with the requirement that each player's strategy be a best response to the other players' strategies. Nash equilibrium does not always make sensible predictions when applied to the extensive form. For instance, reconsider the advertising example discussed above in Table 3. Suppose that firm 1 makes its decision of whether to advertise before firm 2 does, and that firm 2 learns firm 1's choice before it chooses. This scenario is represented in the game tree pictured in Figure 2.

To apply the Nash equilibrium concept to this extensive form game, we must specify what each player does at each node. There are two Nash equilibria of this game in pure strategies. The first is where firm 1 advertises, and firm 2 does not (and firm 2's strategy conditional on firm 1 not advertising is to advertise). The other equilibrium corresponds to the one identified in the normal form: firm 1 does not advertise, and firm 2 advertises regardless of what firm 1 does. This is an equilibrium, since neither wants to change its behavior, given the other's strategy. However, it is not really credible in the following sense: it involves firm 2 advertising even after it has seen that firm 1 has advertised, and even though this action is not in firm 2's interest in that contingency.

To capture the idea that each player's strategy has to be credible, we can solve the game backward. That is, we can look at each decision node that has no successor, and start by making predictions at those nodes. Given those decisions, we can roll the game backward

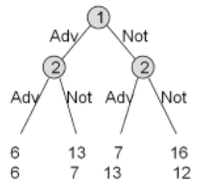


Figure 2: Advertising Choices of Two Competitors

and decide how player's will act at next-to-last decision nodes, anticipating the actions at the last decision nodes, and then iterate. This is called *backward induction*. Consider the choice of firm 2, given that firm 1 has decided not to advertise. In this case, firm 2 will choose to advertise, since 13 is larger than 12. Next, consider the choice of firm 2, given that firm 1 has decided to advertise. In this case, firm 2 will choose not to advertise, since 7 is larger than 6. Now we can collapse the tree. Firm 1 will predict that if it does not advertise, then firm 2 will advertise, while if firm 1 advertises then firm 2 will not. Thus when making its choice, firm 1 anticipates a payoff of 7 if it chooses not to advertise and 13 if it chooses to advertise. Its optimal choice is to advertise. The backward induction prediction about the actions that will be taken is for firm 1 to advertise and firm 2 not to.

Note that this prediction differs from that in the simultaneous move game we analyzed before. Firm 1 has gained a first-mover advantage in the sequential version. Not advertising is no longer a dominant strategy for firm 1, since firm 2's decision depends on what firm 1 does. By committing to advertising, firm 1 forces firm 2 to choose not to advertise. Firm 1 is better off being able to commit to advertising in advance.

A solution concept that capture found in this game and applies to more general classes of

games is known as *subgame perfect equilibrium* (due to Reinhard Selten (1975) Reexamination of the Perfectness Concept for Equilibrium Points in Extensive Games, International Journal of Game Theory 4:25 - 55). A subgame in terms of a finite game tree is simply the subtree that one obtains starting from some given node. Subgame perfection requires that the stated strategies constitute a Nash equilibrium in every subgame (including those with only one move left). So it requires that if we start at any node, then the strategy taken at that node must be optimal in response to the remaining specification of strategies. In the game between the two firms, it requires that firm 2 choose an optimal response in the subgame following a choice by firm 1 to advertise, and so it coincides with the backward induction solution for such a game.

It is worth noting that moving first is not always advantageous. Sometimes it allows one to commit to strategies which would otherwise be untenable, which can be advantageous; but in other cases it may be that the information that the second mover gains from knowing which strategy the first mover has chosen is a more important consideration. For example, suppose that the matching pennies game we discussed above were to be played sequentially so that the kicker had to kick first and the goalie had time to see the kicker's action and then to react and could jump left or right to match the kicker's choice: the advantage would certainly then tip towards the goalie.

This concludes our whirlwind tour of some of the basic tools of game theory. There are many important subjects that I have not touched upon here, including analyses that incorporate incomplete information, repeated games, and behavioral game theory. However, this should provide you with some feeling for a few of the most prominent concepts, and some of the approaches that form the backbone of game theoretic analyses.

3 Some Exercises

Exercise 1 *Product Choices.*

Two electronics firms are making product development decisions. Each firm is choosing between the development of two alternative computer chips. One system has higher efficiency, but will require a larger investment and will be more costly to produce. Based on estimates

of development costs, production costs, and demand, the following present value calculations represent the value of the alternatives (high efficiency chips or low efficiency chips) to the firms.

Table 9: A production-choice game

		Firm 2	
		High	Low
Firm 1	High	1, 2	4, 5
	Low	2, 7	5, 3

The first entry in each box is the present value to firm 1 and the second entry is the present value to firm 2. The payoffs in the above table are not symmetric. Firm 2 has a cost advantage in producing the higher efficiency chip, while firm 1 has a cost advantage in producing the lower efficiency chip. Overall profits are largest when the firms choose different chips and do not compete head to head.

(a) Firm 1 has a dominant strategy. What is it?

(b) Given your answer to part a), what should firm 2 expect firm 1's choice to be? What is firm 2's optimal choice given what it anticipates firm 1 to do?

(c) Do firm 1's strategy (answer to (a)) and firm 2's strategy (answer to (b)) form an equilibrium? Explain.

(d) Compared to (c), firm 1 would make larger profits if the choices were reversed. Why don't those strategies form an equilibrium?

(e) Suppose that firm 1 can commit to a product before firm 2. Draw the corresponding game tree and describe the backward induction/subgame perfect equilibrium.

Exercise **2** *Hotelling's Hotels.*

Two hotels are considering a location along a newly constructed highway through the desert. The highway is 500 miles long with an exit every 50 miles (including both ends). The hotels may choose to to locate at any exit. These will be the only hotels for any traveler using the highway. Each traveler has their own most preferred location along the highway (at some exit) for a hotel, and will choose to go the hotel closest to that location. Travelers most preferred locations are distributed evenly, so that each exit has the same number of travelers who prefer that exit. If both hotels are the same distance from a traveler's most preferred location, then that traveler flips a coin to determine which hotel to stay at. A hotel would each like to maximize the number of travelers who stay at it.

If Hotel 1 locates at the 100 mile exit, where should Hotel 2 locate?

Given Hotel 2's location that you just found, where would Hotel 1 prefer to locate?

Which pairs of locations form Nash equilibria?

Exercise **3** *Backward Induction.*

Find the backward induction solution to Figure 1 and argue that there is a unique subgame perfect equilibrium. Provide a Nash equilibrium of that game that is not subgame perfect.

Exercise **4** *The Colonel Blotto Game.*

Two armies are fighting a war. There are three battlefields. Each army consists of 6 units. The armies must each decide how many units to place on each battlefield. They do this without knowing how many units the other army has committed to a given battlefield. The army who has the most units on a given battlefield, wins that battle, and the army that wins the most battles wins the war. If the armies each have the same number of units on a given battlefield then there is an equal chance that either army wins that battle. A pure strategy for an army is a list $\langle u_1, u_2, u_3 \rangle$ of the number of units it places on battlefields 1, 2, and 3 respectively, where each u_k is in $\{0, 1, \ldots, 6\}$ and the sum of the u_k's is 6. For example, if army A allocates its units (3,2,1), and army B allocates its units (0,3,3), then army A wins the first battle, and army B wins the second and third battles and army B wins the war.

Argue that there is no pure strategy Nash equilibrium to this game.

Argue that mixing uniformly at random over all possible configurations of units is not a mixed strategy Nash equilibrium (hint - show that placing all units on one battlefield is an action that an army would not want to choose if the other army is mixing uniformly at random).

Argue that each army mixing with equal probability between (0,3,3), (3,0,3) and (3,3,0) is not an equilibrium.[14]

Exercise **5** *Divide and Choose.*

Two children must split a pie. They are gluttons and each prefers to eat as much of the pie as they can. The parent tells one child to cut the pie into two pieces and then allows the other child to choose which piece to eat. The first child can divide the pie into any multiple of tenths (for example, splitting it into pieces that are 1/10 and 9/10 of the pie, or 2/10 and 8/10, and so forth). Show that there is a unique backward induction solution to this game.

Exercise **6** *Information and Equilibrium.*

Each of two players receives an envelope containing money. The amount of money has been randomly selected to be between 1 and 1000 dollars (inclusive), with each dollar amount equally likely. The random amounts in the two envelopes are drawn independently. After looking in their own envelope, the players have a chance to trade envelopes. That is, they are simultaneously asked if they would like to trade. If they both say "yes," then the envelopes are swapped and they each go home with the new envelope. If either player says "no," then they each go home with their original envelope.

The actions in this game are actually a full list of whether a player says yes or no for each possible amount of money he or she is initially given. To simplify things, let us write down actions in the following more limited form: an action is simply a number between 0 and 1000, meaning that if they get an envelope with more than that number, then they say "no" and otherwise they say "yes".

[14]Finding equilibria to Colonel Blotto games is notoriously difficult. One exists for this particular version, but finding it will take you some time.

So, for instance, if player 1 chooses action "3", then she says "yes" to a trade when her initial envelope has 1 or 2 or 3 dollars, but says "no" if her envelope contains 4 or more dollars.

In a pure or mixed strategy equilibrium is it possible for both players to choose action "1000" with some positive probability?

Suppose that player 2 does not play action "1000", can a best response of player 1 involve any positive probability on the action "1000"?

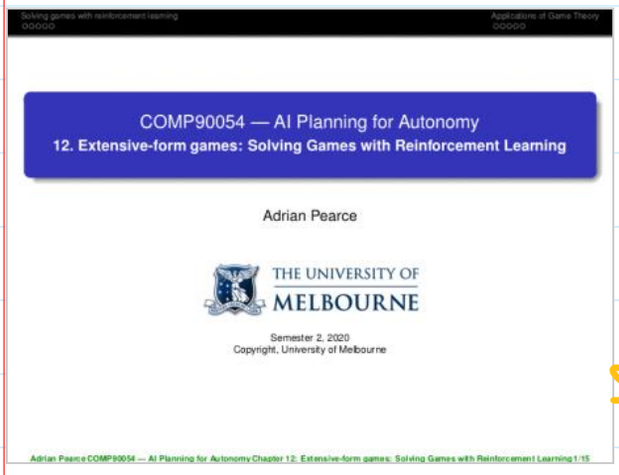
Repeat the above logic to argue that neither player will ever play "999" in an equilibrium.

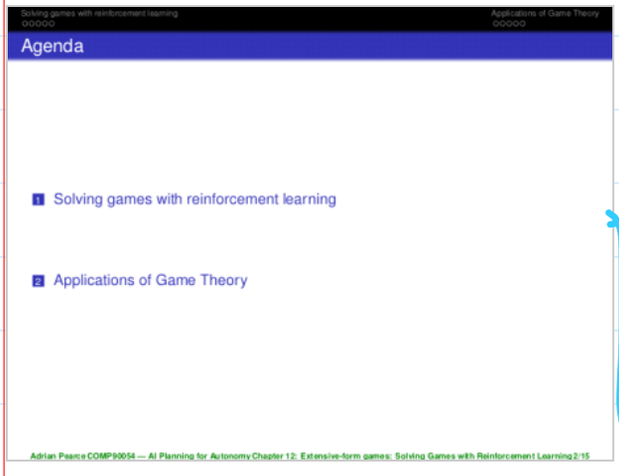
Iterating on this logic, what is the unique Nash equilibrium of this game?

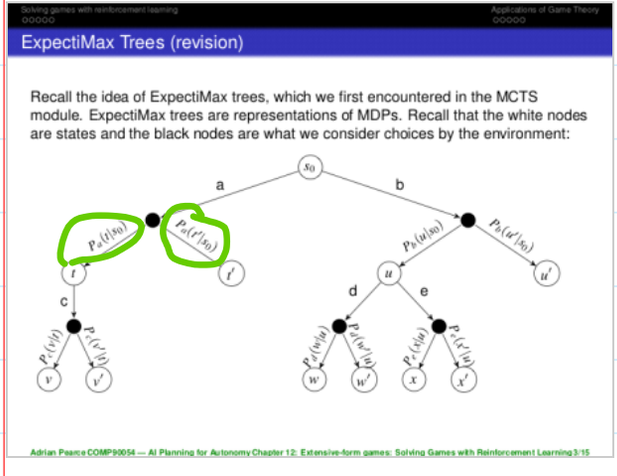
Chapter 14

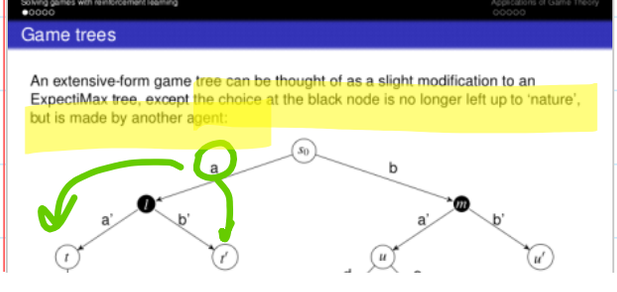
2020年10月22日 星期四 10:10

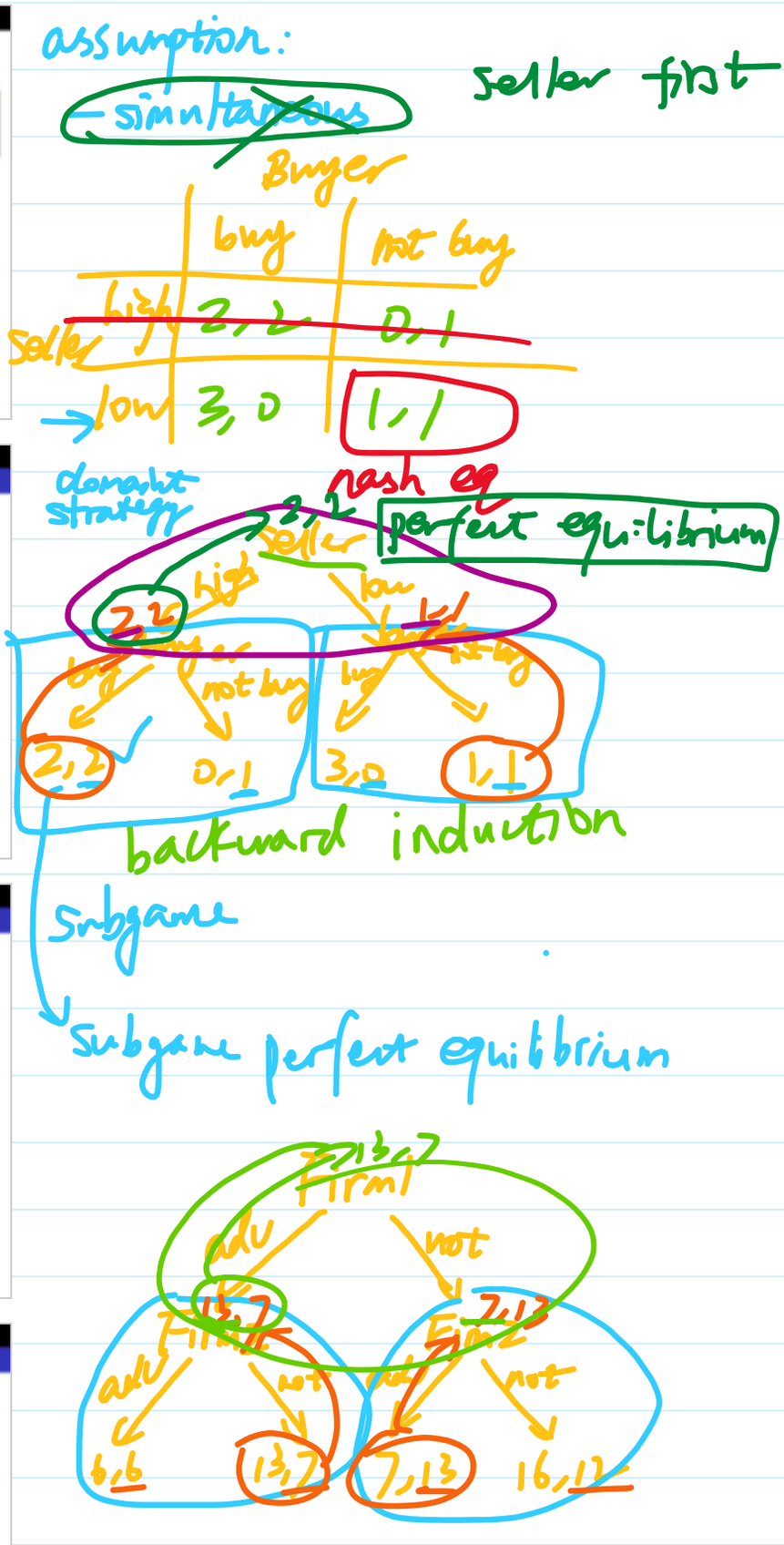
intro12-pr
e-handout





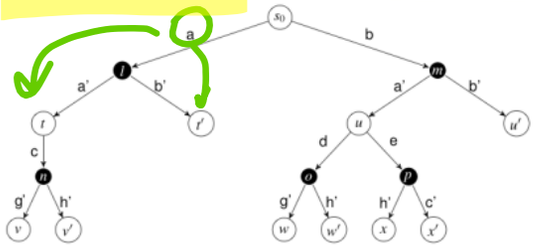






Game trees

An extensive-form game tree can be thought of as a slight modification to an ExpectiMax tree, except the choice at the black node is no longer left up to 'nature', but is made by another agent:



Rewards are only received at the end of games.

MCTS for games

Using MCTS for games is very similar to using MCTS on simulators (week 8).
To use MCTS in games, we simply modify the basic MCTS algorithm as follows:

1. For 'our' moves, we run *selection* as before, however, we also need to select models for our opponents (more on this in the next slide).
2. In the *expansion* step, instead of expanding all child nodes of an action, we run the simulation forward one step. How do we choose the outcome? We simply choose an action for our opponents!
3. We then simulate as before, and we learn the rewards when we receive them.
4. In the *backpropagation* step, instead of using the Bellman equation to calculate the expected return, we simply use the *average* return. If we simulate each step enough times, the average will converge to the expected return.

As with using MCTS for simulation, we have to do repeated simulations for the average to converge.

Choosing actions for opponents

One question that remains is: how should we choose actions for opponents in the selection and expansion step?s There are a few ways to do this.

1. *Random*: Select a random action. This is easy, but it means that we may end up exploring a lot of actions that will never be taken by a good opponent.
2. *Using a fixed policy*: Hand-code a simple stochastic policy that gives reasonable behaviour of the opponent.
3. *Simultaneously learn a policy*: We *learn* a policy for ourselves and our opponents, and we choose actions for our opponents based on the learnt policies.

Simultaneously learning opponent policies

If we want to learn policies for our opponents, we turn this into a *multi-agent reinforcement learning problem*. We are not going to go into the details of multi-agent reinforcement learning in this subject, but to give some intuition that may help on your final project:

1. The payoffs for each agent are their own reward function.
2. We learn n Q-functions — one for each of the n players. The function $Q_i(s, a)$ is the Q-function for agent i.
3. When we have a selection or expansion in the tree, we use the Q-function for the respective agent.
4. Similarly, when we back propagate values back, we update the right agent at the right node.

Model-free reinforcement learning

Similarly, we can apply model-free techniques for multi-agent problems. In a truly model-free problem, we cannot simulate our opponent necessarily, however, our opponent will be executing their own moves, so we don't need to.

A simple way to use model-free techniques in a multi-agent setting is simply to delay the update until our opponents moves have been selected.

If we consider the Q-learning algorithm, the inner loop is:

Choose a from s using policy derived from Q (e.g. ϵ-greedy)
Take action a, observe r [, observe s']
Wait until opponents have executed and observe final outcome s'
$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

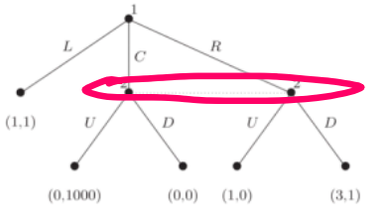
Here, the light grey text is removed from the original Q-learning algorithm, and the bold is added.

Poker

Image from
https://www.cs.cmu.edu/~sandholm/Solving%20games.Science-2015.pdf

Poker — Imperfect Information Games

Imperfect information games are a generalisation of extensive-form games in which players do not necessarily know what moves their opponent has made in the past, so when we select an action we are unsure which part of the tree we are in. For example, the following game tree, player 2 does not see whether player 1 has played C or R. The dotted line indicates cannot tell the difference between the two states:



Poker — Scaling with Abstraction

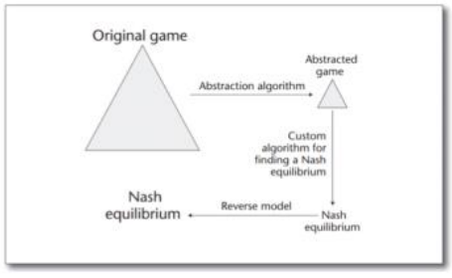


Image from
https://www.cs.cmu.edu/~sandholm/Solving%20games.Science-2015.pdf

Security games — Bayesian Stackelberg Games

A Bayesian game is one in which the is incomplete information about the game; in particular, we do not know the pay-offs of our opponent. Instead, we know about different *types* of player. For example, we can model uncertainty in our opponent's pay-off by modelling that there are two possible attackers with differing pay-offs:

Adversary (P=0.3)

Defender	T1	T2
T1	5, -3	-1, 1
T2	-5, 5	2, -1

Adversary (P=0.7)

Defender	T1	T2
T1	5, -2	-1, 3
T2	-5, 2	2, -2

From this, we can assign e.g. a probability of 0.3 to attacker type 1 and 0.7 to attacker type 2. The problem is then to select a strategy that maximises expected utility over both attackers.

Security games – PAWS: Stopping animal poaching

Check out this brilliant application of game theory (and other AI techniques) to help prevent animal poaching by using mixed strategies to randomise strategies about where to place anti-poaching officers:

Save the Wildlife, Save the Planet: Protection Assistant for Wildlife Security (PAWS)

Or go to the URL: `https://youtu.be/ai6yhbx5iGw?t=94`

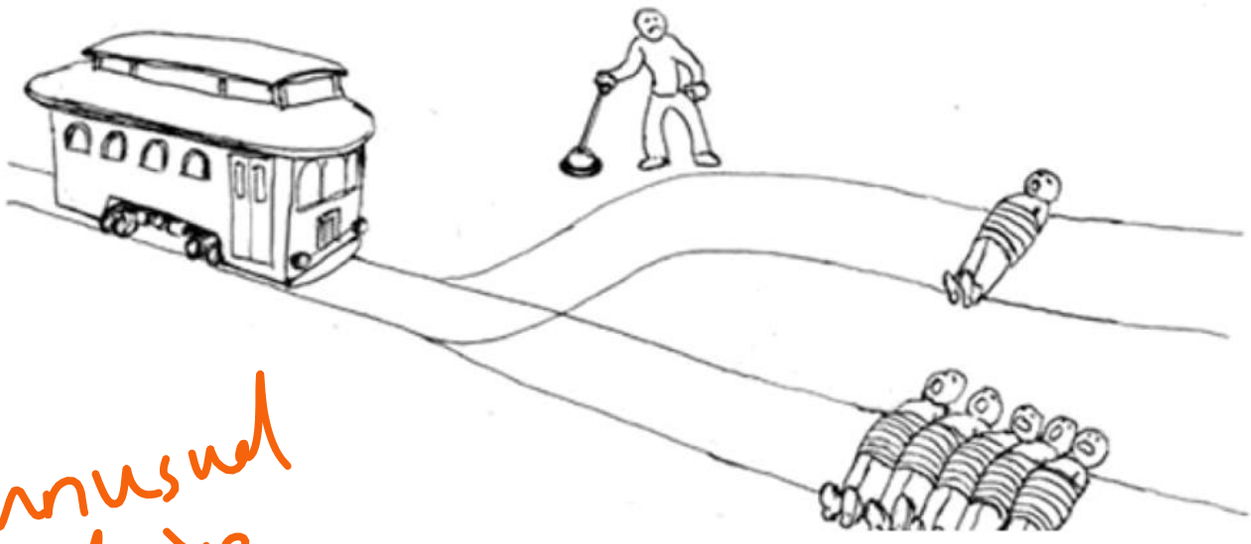
Chapter 15

2020年10月30日 星期五 07:13

intro13-pr
e-handout

Image source: https://www.google.com/selfdrivingcar/

The Trolley Problem



unusual choice

Image source: http://www.relativelyinteresting.com/the-trolley-problem-a-thought-experiment-that-tests-our-morality/

Ethics

Utilitarianistic ethics: only outcomes matter, and we should always opt for the best.

Deontological ethics: actions matter too! We have a duty to help others as well as a duty not to harm others, but the latter is stronger than the former.

What should we do?

The 'Fat Man' Trolley Problem

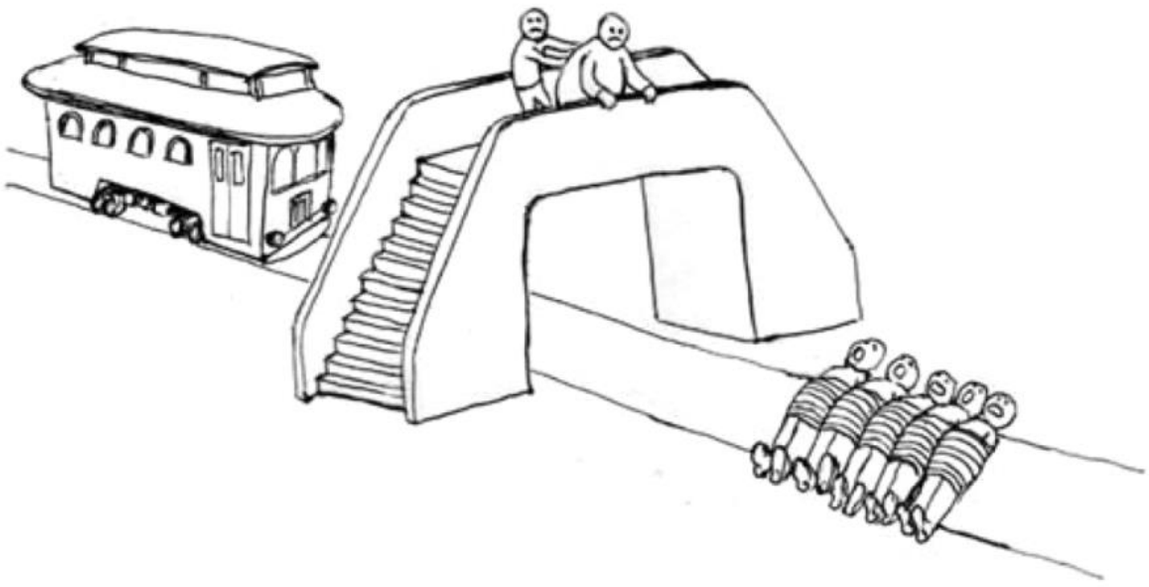
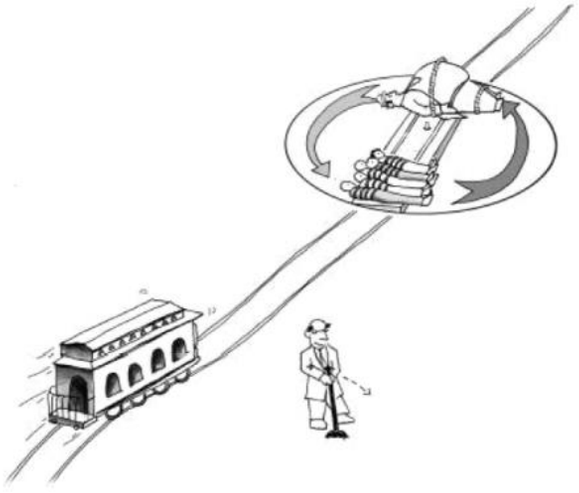


Image source: http://www.relativelyinteresting.com/the-trolley-problem-a-thought-experiment-that-tests-our-morality/

The ‘Fat Man’ Trolley Problem (Lazy Susan)



In Lazy Susan you can save the five by revolving the turntable 180 degrees—this will have the unfortunate consequence of placing one man directly in the path of the train. Should you rotate the Lazy Susan?

(Figure: Would you kill the Fat Man?)

Many variants of the Trolley Problem

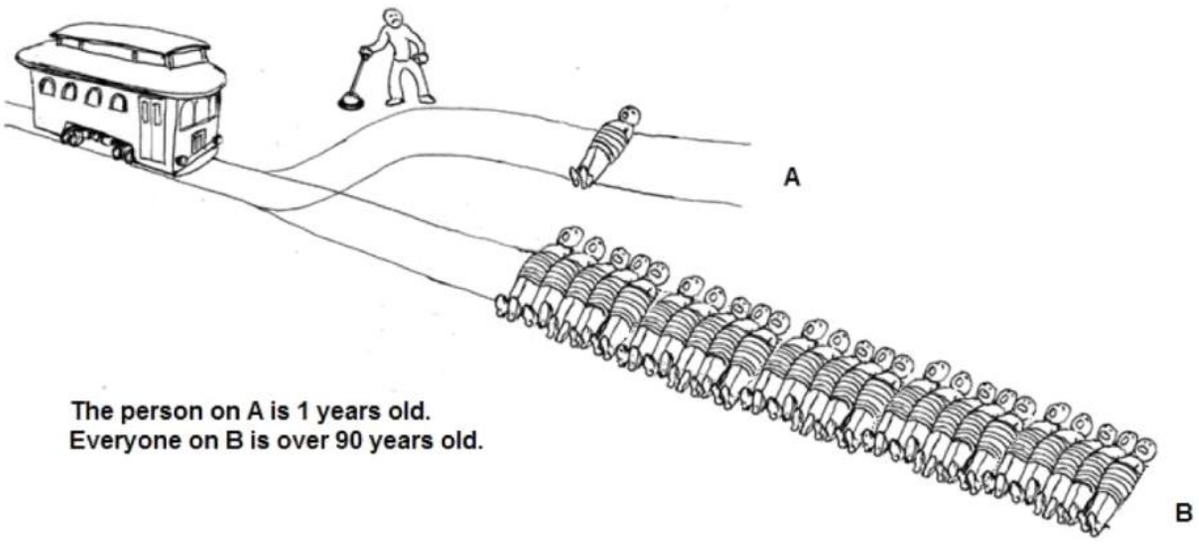
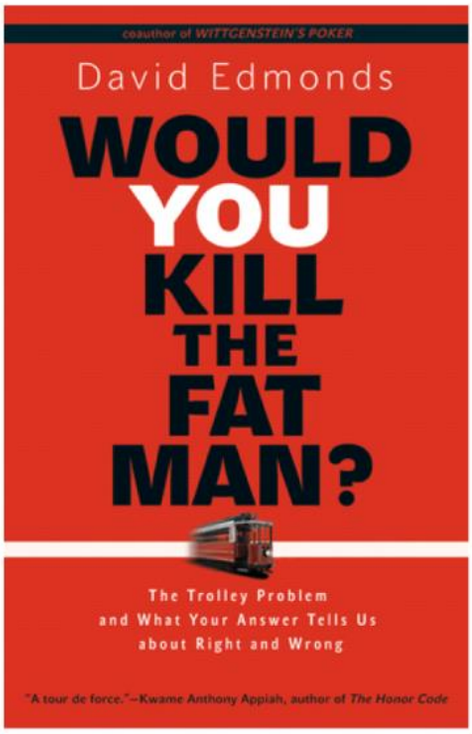
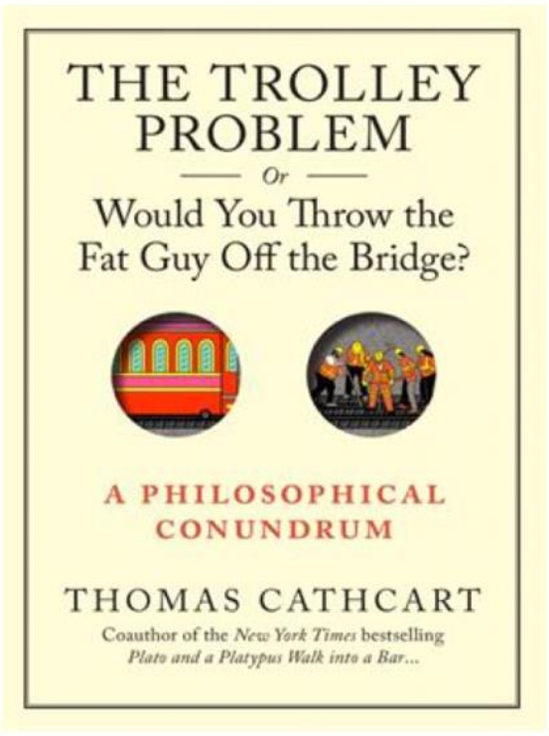


Image source: http://www.relativelyinteresting.com/the-trolley-problem-a-thought-experiment-that-tests-our-morality/

Experimental ethics





Pre-programmed ethics: The case of autonomous vehicles

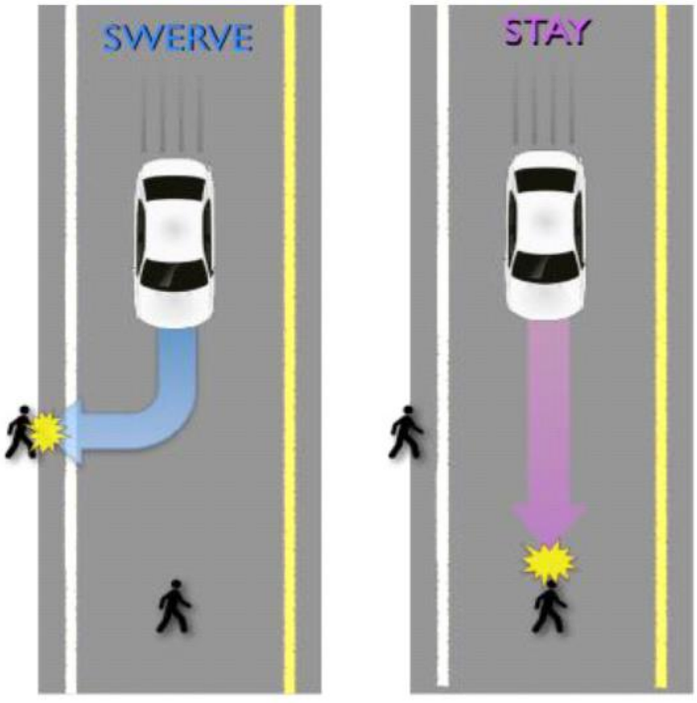
Split-second driver decision:

- reckless driver → criminal liability
- otherwise → blame "fate", "bad luck", "heat of the moment", and do not blame the driver

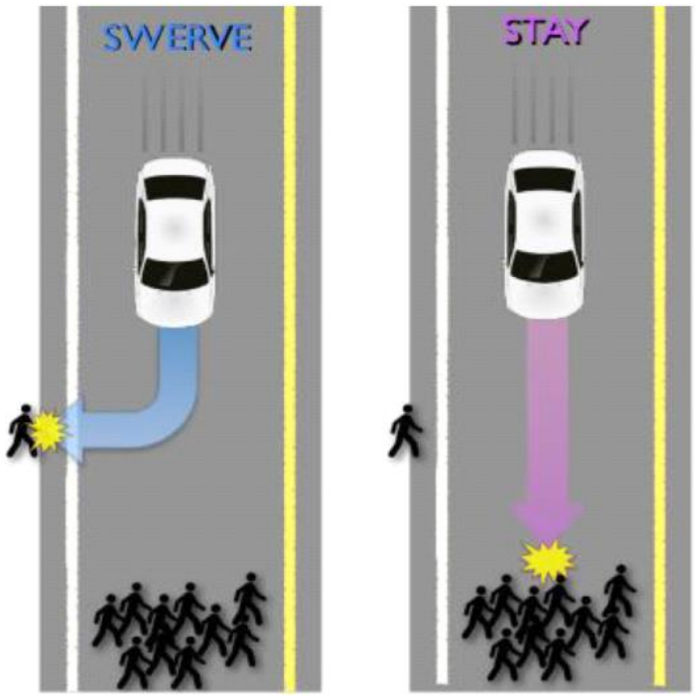
Automated decision-making:

- We must decide in advance! No split-second decisions
- What behaviour should be programmed?

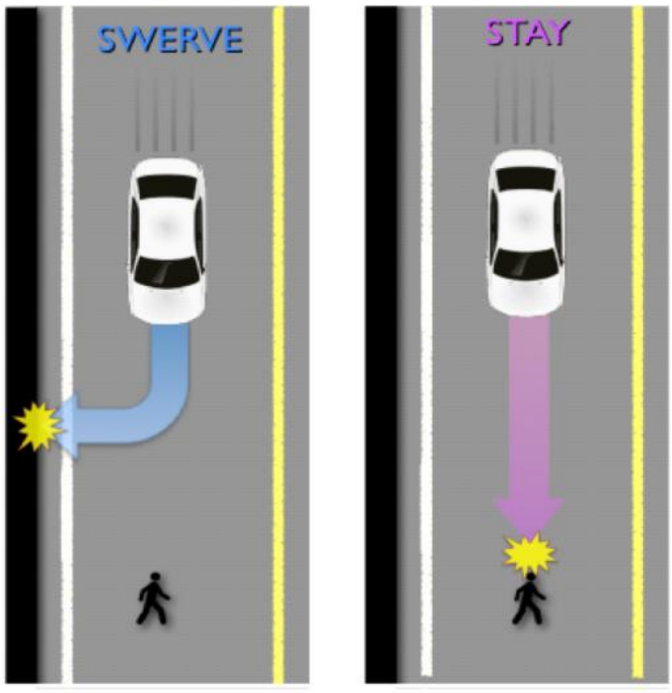
Swerve or stay?



Swerve or stay?



Swerve or stay?



In some larger-scale experiments

Would you buy a self-driving car that protected passengers instead of the driver?

Despite preferring self-sacrificing vehicles ...

... respondents were less likely to buy such cars.

Trade-offs must be made by law-makers to solve this *social dilemma*

But.....

What about related but different scenarios:

- 99.95% confidence an object is not a person
- What if the cars knows there is a child on board?
- Buyer liability: If buyers can choose the “moral algorithm” are they liable for its choices?

Fairness

In **data mining** and **computer vision**, a machine learning algorithm is said to be fair, or to have fairness, if its results are independent of given variables, especially those considered sensitive, such as the traits of individuals which should not correlate with the outcome

- For example gender, ethnicity, etc. (there are well-known examples of unfair ML implementations, e.g. law enforcement and recruitment)

Here is a simple (although not machine learning) example demonstrating why we must take care of data distributions when developing (any) algorithms:

- In 2020, UK dropped its (COVID-19 motivated) exam-grading policy for General Certificate of Secondary Education (GCSE) results. You can watch this explainer (posted by the Financial Times) **What went wrong with the A-level algorithm?**
  - https://www.youtube.com/watch?v=jHtMLEhDOVE

Fairness in AI planning & reinforcement learning

What about learning behaviour for augmented (human-agent) intelligence or fully autonomous systems?

- Model-free AI planning and reinforcement learning **is potentially subject to the same fairness limitations** as in computer vision and data mining, due to for example
  - sampling complexity (ability to sample enough data and/or query simulators to generate meaningful probability distributions)
  - non-stationarity (when the future is not symmetrical with the past independent of the amount of data sampled – adversarial and competitive games are typical examples)
- model-based approaches rely on the correctness/completeness of the *modelling language* encoding

Who should decide?

AI programmers?

Companies?

Lawmakers?

The public?

Ethicists and philosophers?

The Moral Machine at MIT Media Lab

http://moralmachine.mit.edu/

Reading

(Chapter 5: Fat man, Loop and Lazy Susan) *Would you kill the fat man?* by David Edmonds, 2013

https://ebookcentral.proquest.com/lib/unimelb/detail.action?docID=1275331

(External link, UniMelb login required to access e-book)

THE UNIVERSITY OF MELBOURNE

Thank you