FAI LAB 4
Beyond Classical Search

Paolo Morettin

2024-25

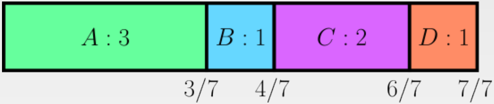
Code available

You can find the **code** of the lab sessions on GitHub:

`https://github.com/paolomorettin/FAI-code`

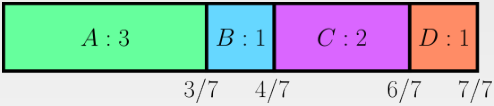
On non-determinism

- In a computer: (seed number $+$) pseudo-random algorithms
- We need to agree on a **weighted sampling** procedure



On non-determinism

- In a computer: (seed number $+$) pseudo-random algorithms
- We need to agree on a **weighted sampling** procedure



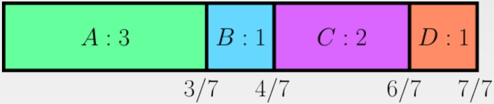
- The idea: a list of pre-determined choices $\in [0, 1]$ is provided

$$\mathit{choices} = [0.5, 0.1, 0.9]$$

Selected: *B*

On non-determinism

- In a computer: (seed number $+$) pseudo-random algorithms
- We need to agree on a **weighted sampling** procedure



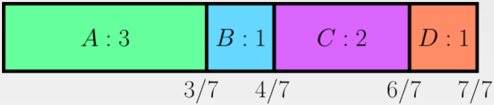
- The idea: a list of pre-determined choices $\in [0, 1]$ is provided

$$\mathit{choices} = [0.5, 0.1, 0.9]$$

Selected: *A*

On non-determinism

- In a computer: (seed number $+$) pseudo-random algorithms
- We need to agree on a **weighted sampling** procedure



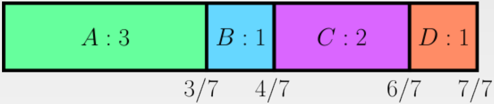
- The idea: a list of pre-determined choices $\in [0, 1]$ is provided

$$\textit{choices} = [0.5, 0.1, 0.9]$$

Selected: D

On non-determinism

- In a computer: (seed number $+$) pseudo-random algorithms
- We need to agree on a **weighted sampling** procedure



- The idea: a list of pre-determined choices $\in [0,1]$ is provided

$$\mathit{choices} = [0.5, 0.1, 0.9]$$

Selected: *D*

- Then we cycle

On non-determinism

```python
def select_choice(options, choices):
    '''Simulated stochastic process that deterministically pick an
    option given a pre-determined list of choices. Choices are
    cycled through. Options are weighted.

    '''
    choice = choices.pop(0)
    choices.append(choice) # cycling

    w_sum = sum(w for _, w in options)
    p = 0
    for opt, w in options:
        p += w/w_sum
        if choice <= p:
            return opt, p
```

Local search: hill climbing

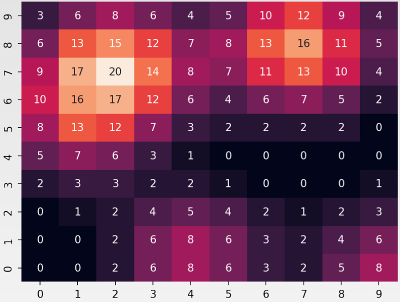
- Goal: find s^* maximizing an objective function $f(.)$
- One rule: **do not look down**

$$\text{candidates } H(curr) = \{s \in Neigh(curr) \mid f(s) > f(curr)\}$$

- Different variants:
  - **steepest**: $next = argmax_{n \in H(curr)} f(n)$
  - **stochastic (unweighted)**: $next \sim \mathcal{U}(H(curr))$
  - **stochastic**: $next \sim p(n) \propto f(H(curr))$
- Multiple (parallel) restarts of the above

Non-convex optimization on a 2D grid

- **Goal**: maximize a randomly generated function $f(x, y)$



- *Actions*$((x, y))$ results in the following order of next states:

$$[(x-1, y-1), (x, y-1), (x+1, y-1),$$
$$(x-1, y), (x+1, y),$$
$$(x-1, y+1), (x, y+1), (x+1, y+1)]$$

6	7	8
4		5
1	2	3

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet
- At each iteration, a **population** is considered:

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet
- At each iteration, a **population** is considered:
  - Individuals are selected for reproduction (with p proportional to f)

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet
- At each iteration, a **population** is considered:
  - Individuals are selected for reproduction (with p proportional to f)
  - The offspring is the result of the combination of the parents

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet
- At each iteration, a **population** is considered:
  - Individuals are selected for reproduction (with p proportional to f)
  - The offspring is the result of the combination of the parents
  - Mutations can happen with small probability

Local search: genetic algorithms

- Well-known family of **evolutionary algorithms**
- The objective function f is called *fitness*
- Each solution is encoded as a string over a finite alphabet
- At each iteration, a **population** is considered:
  - Individuals are selected for reproduction (with p proportional to f)
  - The offspring is the result of the combination of the parents
  - Mutations can happen with small probability
  - Elitism: retain best scoring individuals from the previous generation, cull the weak f monotonically increases

GA example: the quest for the Master Sandwitch

GA example: the quest for the Master Sandwitch

- The **DNA** of a sandwitch:

main	*side*	*sauce*	*bread*
Ham	**L**ettuce	**M**ayo	**B**un
Salami	**T**omato	**Y**oghurt	**W**rap
Falafel	**O**nions	**G**arlic	**P**ita
Kebab	**B**ell peppers		

(e.g. HLMP = a ham / lettuce / mayo / pita sandwich)

GA example: the quest for the Master Sandwich

- The **DNA** of a sandwich:

main	*side*	*sauce*	*bread*
Ham	**L**ettuce	**M**ayo	**B**un
Salami	**T**omato	**Y**oghurt	**W**rap
Falafel	**O**nions	**G**arlic	**P**ita
Kebab	**B**ell peppers		

(e.g. HLMP = a ham / lettuce / mayo / pita sandwich)

- Let's assume a simple fitness function:

$$f(x) = \begin{cases} 0 & \text{if } x_0 = H \\ 1 & \text{if } x_0 = S \\ 2 & \text{if } x_0 = F \\ 3 & \text{if } x_0 = K \end{cases} + \begin{cases} 0 & \text{if } x_1 = L \\ 1 & \text{if } x_1 = T \\ 2 & \text{if } x_1 = O \\ 3 & \text{if } x_1 = B \end{cases} + \begin{cases} 0 & \text{if } x_2 = M \\ 1 & \text{if } x_2 = Y \\ 2 & \text{if } x_2 = G \end{cases} + \begin{cases} 0 & \text{if } x_3 = B \\ 1 & \text{if } x_3 = W \\ 2 & \text{if } x_3 = P \end{cases}$$

(It is an **oversimplification** anyway, no bacon considered)

Online search

Online search

- **Interleaving** *computation* and *action*

Online search

- **Interleaving** *computation* and *action*
- Useful for **dynamic** task environments, but also **nondeterministic**

Online search

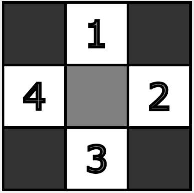
- **Interleaving** *computation* and *action*
- Useful for **dynamic** task environments, but also **nondeterministic**
- Agent knows the following:
  - $Actions(s)$, what can be done
  - $IsGoal(s)$, of course
  - $s' = Result(s, a)$ not known in advance
  - $cost(s, a, s')$ known only after executing a

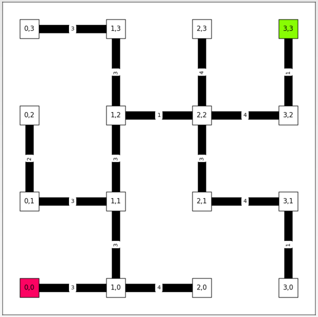
Online search

- **Interleaving** *computation* and *action*
- Useful for **dynamic** task environments, but also **nondeterministic**
- Agent knows the following:
  - $Actions(s)$, what can be done
  - $IsGoal(s)$, of course
  - $s' = Result(s, a)$ not known in advance
  - $cost(s, a, s')$ known only after executing a
- We focus on **safely explorable** state spaces
  a goal state is reachable from any reachable state (no dead-ends)

Maze problems

- **Goal**: reach the green room
- Order of $Actions(s) = [N, E, S, W]$





- undirected graph $\rightarrow$ reversible actions
- $cost(s, a, s') = cost(s', a', s)$

Online DFS

function ONLINE-DFS-AGENT(*problem*, s') **returns** an action
  persistent: s, a, the previous state and action, initially null
  result, a table mapping $(s,\ a)$ to s', initially empty
  untried, a table mapping s to a list of untried actions
  unbacktracked, a table mapping s to a list of states never backtracked to

```
if problem.IS-GOAL(s') then return stop
if s' is a new state (not in untried) then untried[s'] ← problem.ACTIONS(s')
if s is not null then
    result[s, a] ← s'
    add s to the front of unbacktracked[s']
if untried[s'] is empty then
    if unbacktracked[s'] is empty then return stop
    else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
else a ← POP(untried[s'])
s ← s'
return a
```

Learning Real Time A*

function LRTA*-AGENT($problem$, s', h) **returns** an action
  s, a, the previous state and action, initially null
  persistent: $result$, a table mapping (s, a) to s', initially empty
  H, a table mapping s to a cost estimate, initially empty

  if IS-GOAL(s') **then return** *stop*
  if s' is a new state (not in H) **then** $H[s'] \leftarrow h(s')$
  if s is not null **then**
    $result[s, a] \leftarrow s'$
    $H[s] \leftarrow \min_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(s, b, result[s, b], H)$
  $a \leftarrow \operatorname{argmin}_{b \in \text{ACTIONS}(s)} \text{LRTA*-COST}(problem, s', b, result[s', b], H)$
  $s \leftarrow s'$
  return a

function LRTA*-COST($problem$, s, a, s', H) **returns** a cost estimate
  if s' is undefined **then return** $h(s)$
  else return $problem$.ACTION-COST$(s, a, s') + H[s']$