南京大学
人工智能学院
School of Artificial Intelligence, Nanjing University

Lecture 5: Search 4
Bandits and MCTS

Previously...

Path-based search

- Uninformed search
  - Depth-first, breadth first, uniform-cost search
- Informed search
  - Best-first, **A* search**

Adversarial search

- Alpha-Beta search

Beyond classical search

Bandit search

Tree search: Monte-Carlo Tree Search

Functions for pseudo-random numbers

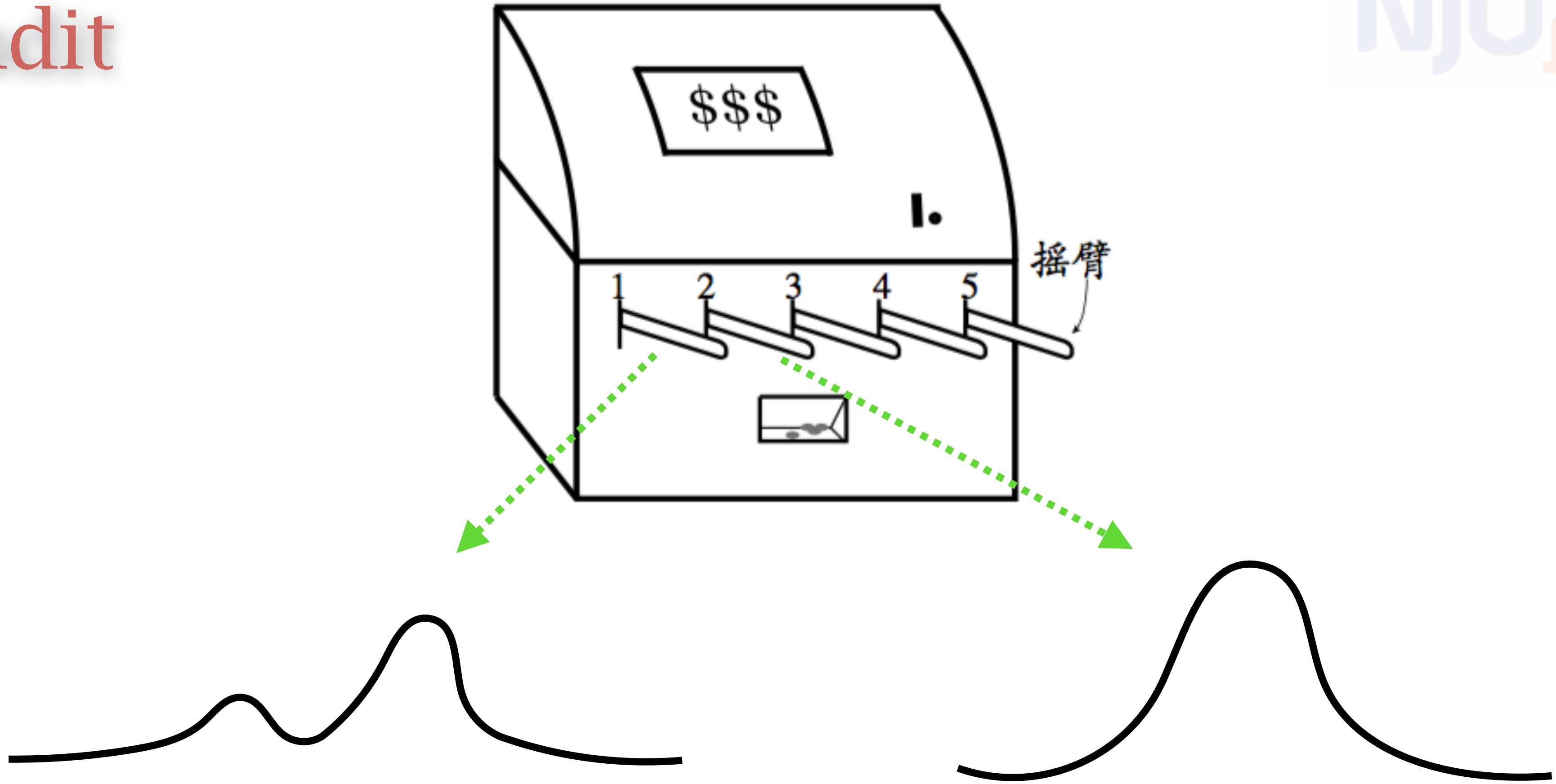
in C++

```
#include <stdlib.h>
srand(seed);
int r = rand();        0~RAND_MAX
```

in JAVA

```
import java.util.Random;
Random rnd = new Random(seed);
int r = rnd.nextInt(upper);        0~upper-1
```

Bandit



Multiple arms
Each arm has an expected reward,
but unknown, with an unknown distribution

Maximize your award in fixed trials

Simplest strategies

Two simplest strategies

Exploration-only:

for T trails and K arms, try each arm T/K times

problem? waste on suboptimal arms

Exploitation-only:

1. try each arm once
2. try the observed best arm T-K times

problem? risk of wrong best arm

ε-greedy

Balance the exploration and exploitation:

with ε probability, try a random arm
with 1-ε probability, try the best arm

ε controls the balance

输入: 摇臂数 K;
奖赏函数 R;
尝试次数 T;
探索概率 ϵ.

过程:

$r = 0$;
$\forall i = 1, 2, \ldots K: \ Q(i) = 0, \ \mathrm{count}(i) = 0$;
for $t = 1, 2, \ldots, T$ **do**
  if $\mathrm{rand}() < \epsilon$ **then**
    $k =$ 从 $1, 2, \ldots, K$ 中以均匀分布随机选取
  else
    $k = \arg\max_i Q(i)$
  end if
  $v = R(k)$;
  $r = r + v$;
  $Q(k) = \frac{Q(k) \times \mathrm{count}(k) + v}{\mathrm{count}(k) + 1}$;
  $\mathrm{count}(k) = \mathrm{count}(k) + 1$;
end for

输出: 累积奖赏 r

Softmax

Balance the exploration and exploitation:

Choose arm with probability

$$P(k) = \frac{e^{\frac{Q(k)}{\tau}}}{\sum_{i=1}^{K} e^{\frac{Q(i)}{\tau}}}, \qquad (16.4)$$

τ controls the balance

输入: 摇臂数 K;
奖赏函数 R;
尝试次数 T;
温度参数 τ.

过程:

$r = 0$;
$\forall i = 1, 2, \ldots K : \ Q(i) = 0, \ \text{count}(i) = 0$;
for $t = 1, 2, \ldots, T$ **do**
    $k =$ 从$1, 2, \ldots, K$ 中根据式(16.4)随机选取
    $v = R(k)$;
    $r = r + v$;
    $Q(k) = \frac{Q(k) \times \text{count}(k) + v}{\text{count}(k) + 1}$;
    $\text{count}(k) = \text{count}(k) + 1$;
end for

输出: 累积奖赏 r

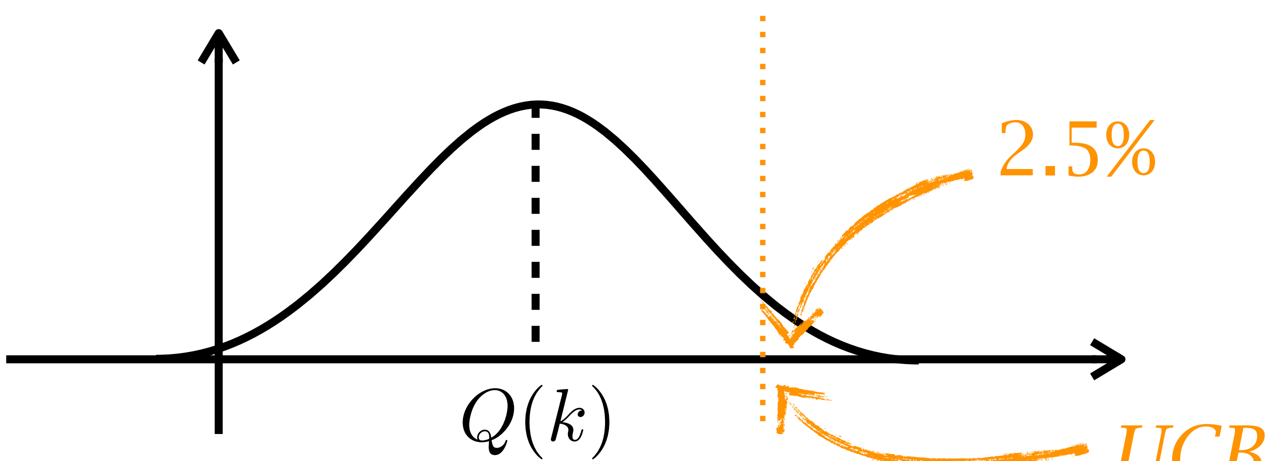
Upper-confidence bound

Balance the exploration and exploitation:

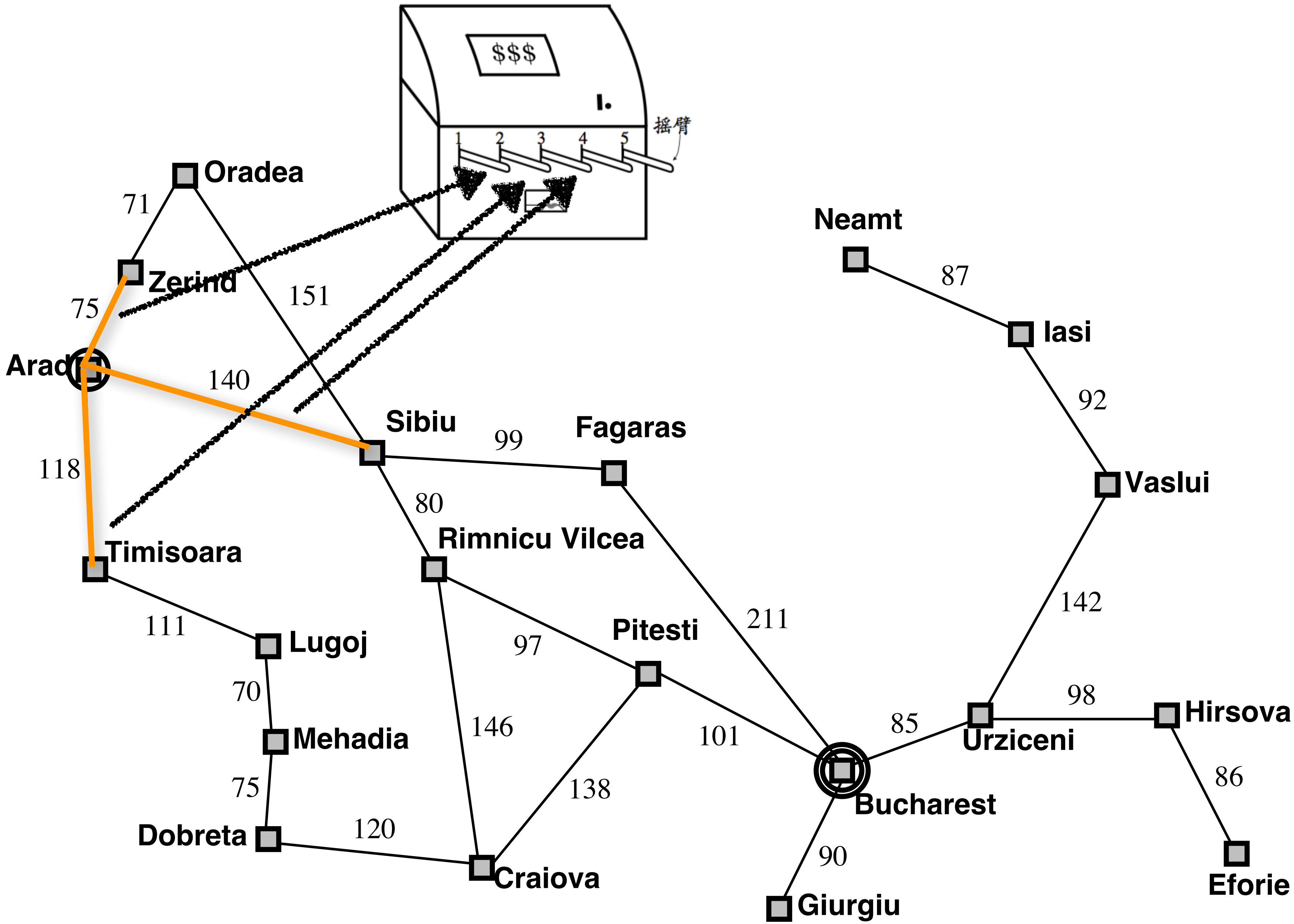
Choose arm with the largest value of

average reward + upper confidence bound

$$Q(k) + \sqrt{\frac{2 \ln n}{n_k}},$$

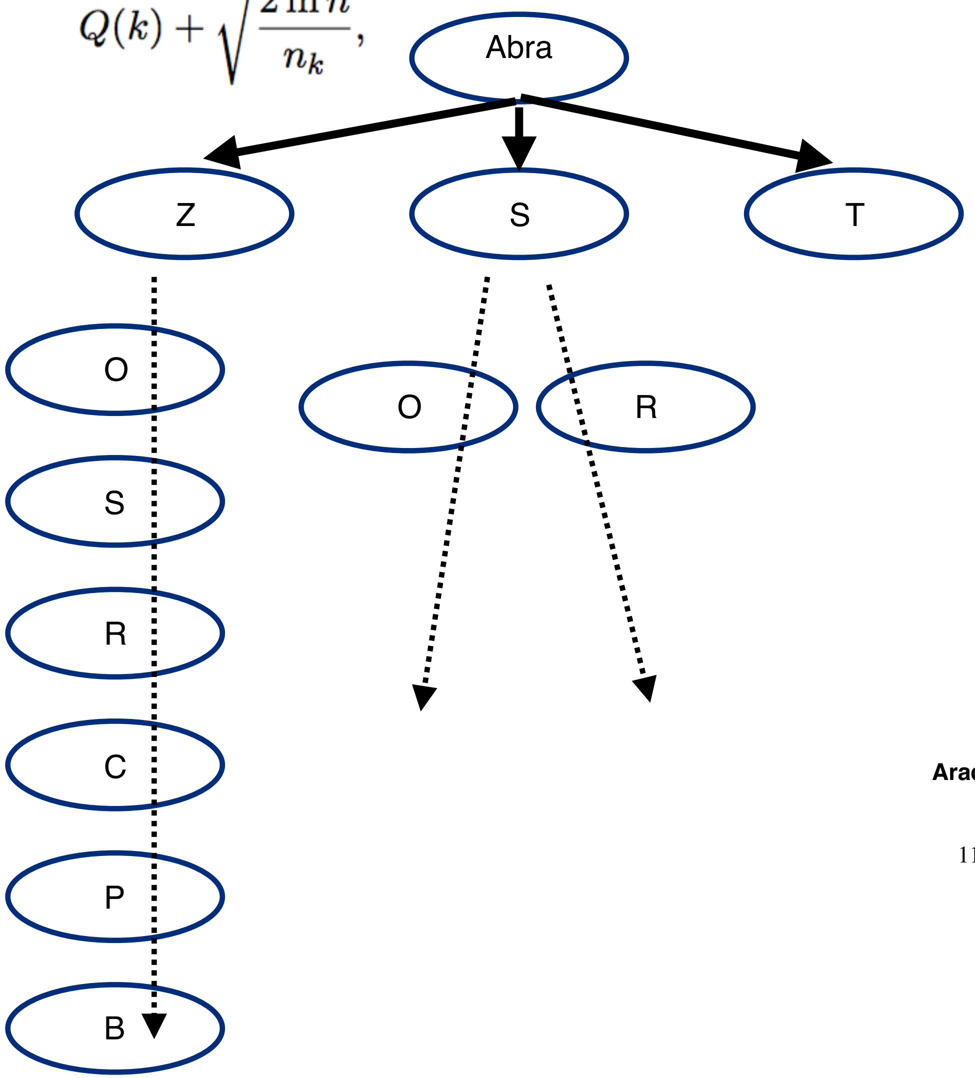


Use bandit to search



Use bandit to search

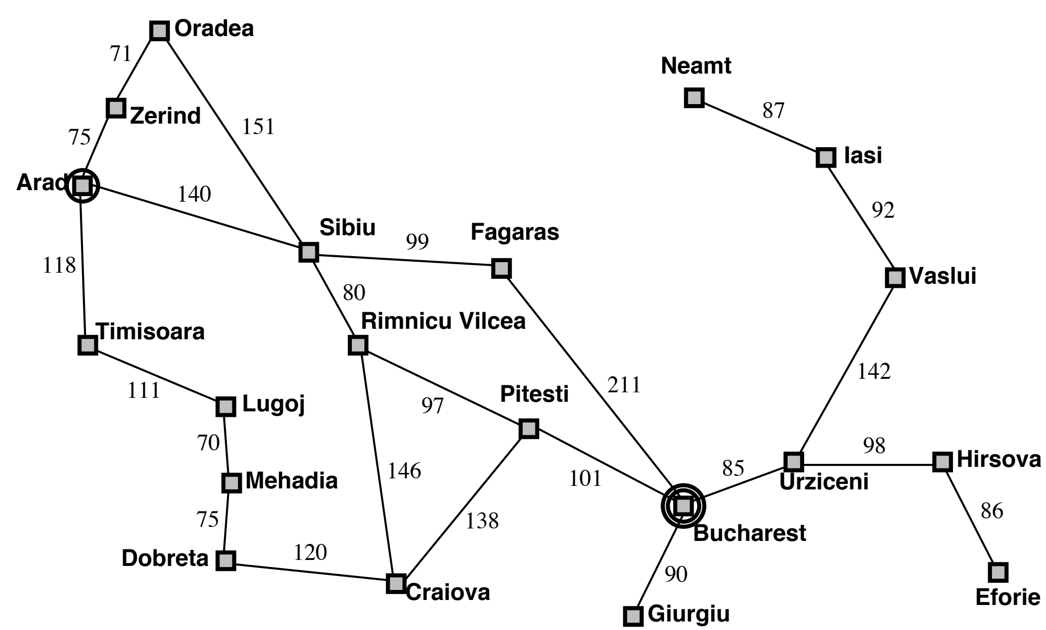
$$Q(k) + \sqrt{\frac{2\ln n}{n_k}},$$



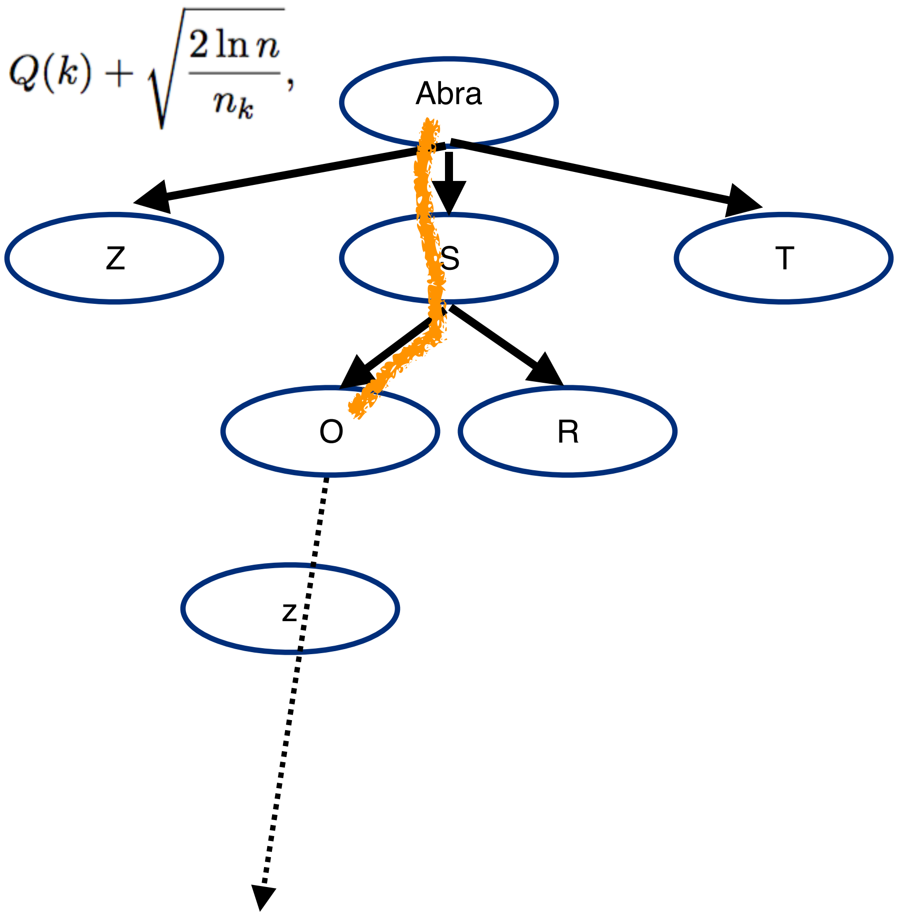
762 a roll-out

use many roll-outs to estimate the average cost of each arm

arm selection: UCB



From bandit to tree



grow a tree

update the values along the path

Monte-Carlo Tree Search

also called Upper-Confidence Tree (UCT)

Kocsis Szepesvári, 06

Gradually grow the search tree:

- Iterate Tree-Walk
  - Building Blocks
    - Select next action

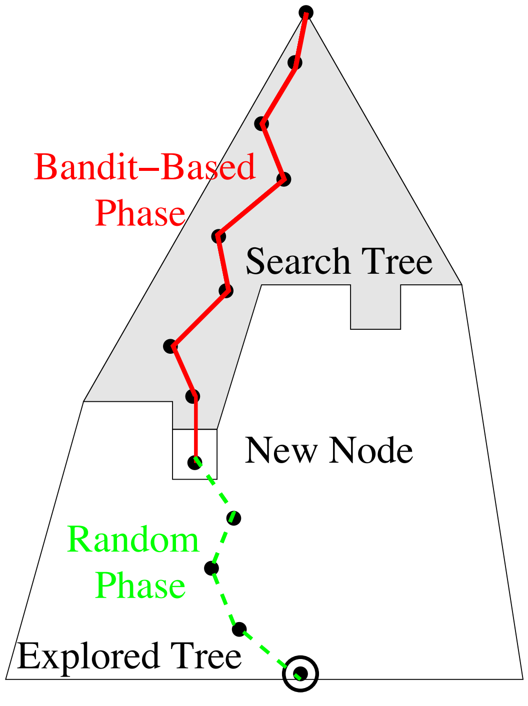
      Bandit phase
    - Add a node

      Grow a leaf of the search tree
    - Select next action bis

      Random phase, roll-out
    - Compute instant reward

      Evaluate
    - Update information in visited nodes

      Propagate
- Returned solution:
  - Path visited most often



Monte-Carlo Tree Search

```
private TreeNode select() {
    TreeNode selected = null;
    double bestValue = Double.MIN_VALUE;
    for (TreeNode c : children) {
        double uctValue = c.totValue / (c.nVisits + epsilon) +
                Math.sqrt(Math.log(nVisits+1) / (c.nVisits + epsilon)) +
                    r.nextDouble() * epsilon;
        // small random number to break ties randomly in unexpanded nodes
        if (uctValue > bestValue) {
            selected = c;
            bestValue = uctValue;
        }
    }
    return selected;
}
```

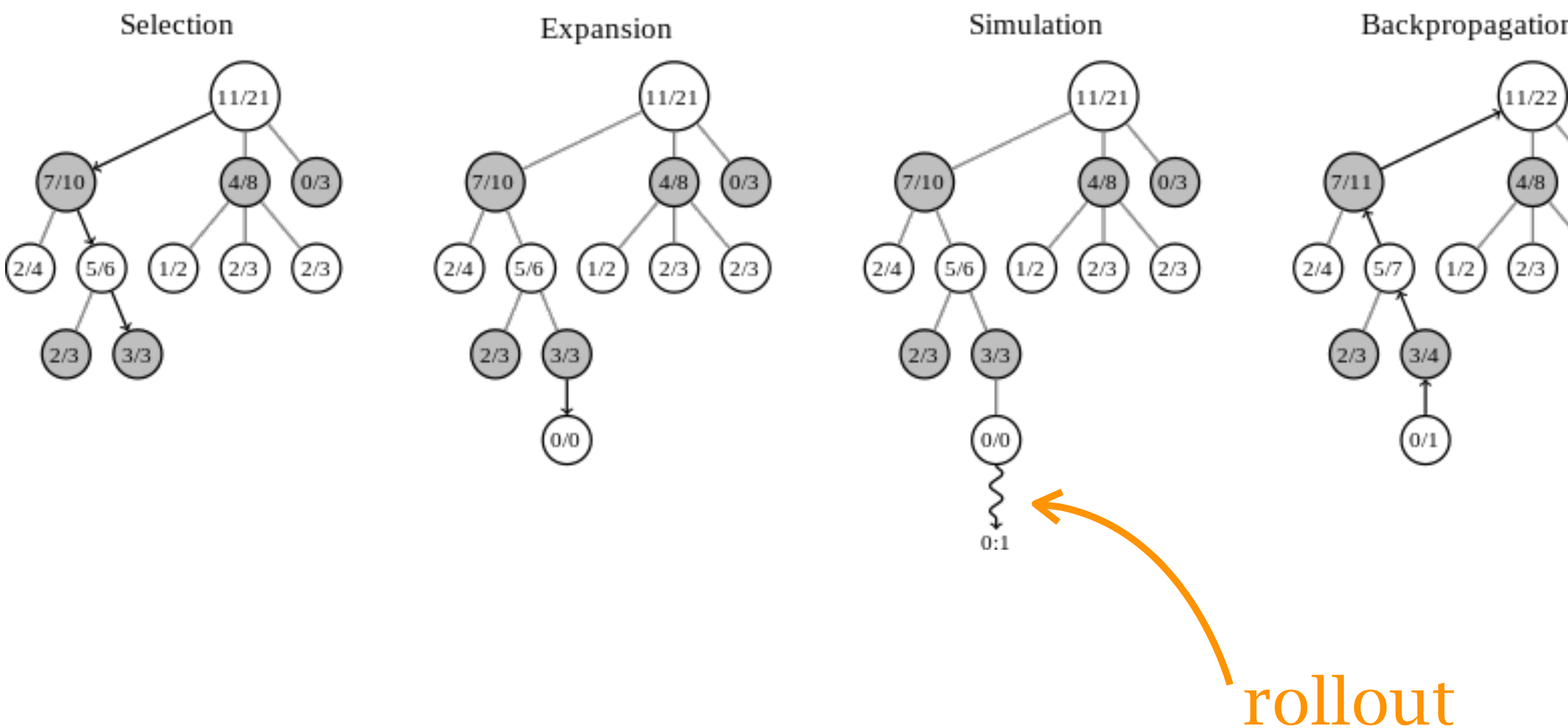
```
    totValue += value;
}
```

```
        cur = cur.select();
        visited.add(cur);
    }
    cur.expand();
    TreeNode newNode = cur.select();
    visited.add(newNode);
    double value = rollOut(newNode);
    for (TreeNode node : visited) {
        // would need extra logic for n-player game
        node.updateStats(value);
    }
}
```

codes from http://mcts.ai/code/java.html

Monte-Carlo Tree Search

Example:



Pic from https://en.wikipedia.org/wiki/Monte_Carlo_tree_search#cite_note-Kocsis-Szepesvari-5

Monte-Carlo Tree Search

optimal? Yes, after infinite tries

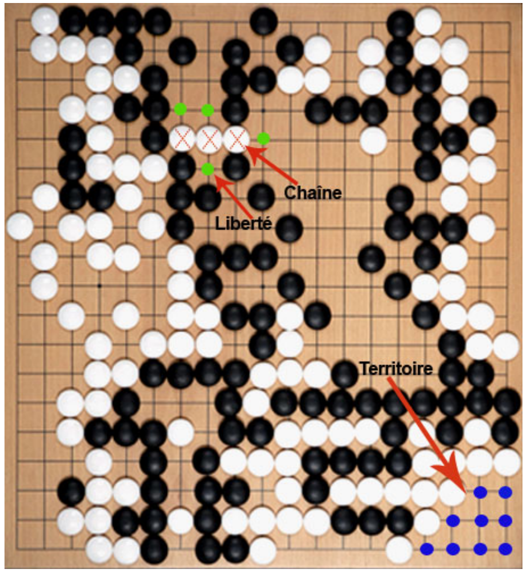
compare with alpha-beta pruning
no need of heuristic function

Monte-Carlo Tree Search

Improving random rollout

Monte-Carlo-based Brügman 93

1. Until the goban is filled,
   add a stone (black or white in turn)
   at a uniformly selected empty position
2. Compute $r = \text{Win(black)}$
3. The outcome of the tree-walk is r

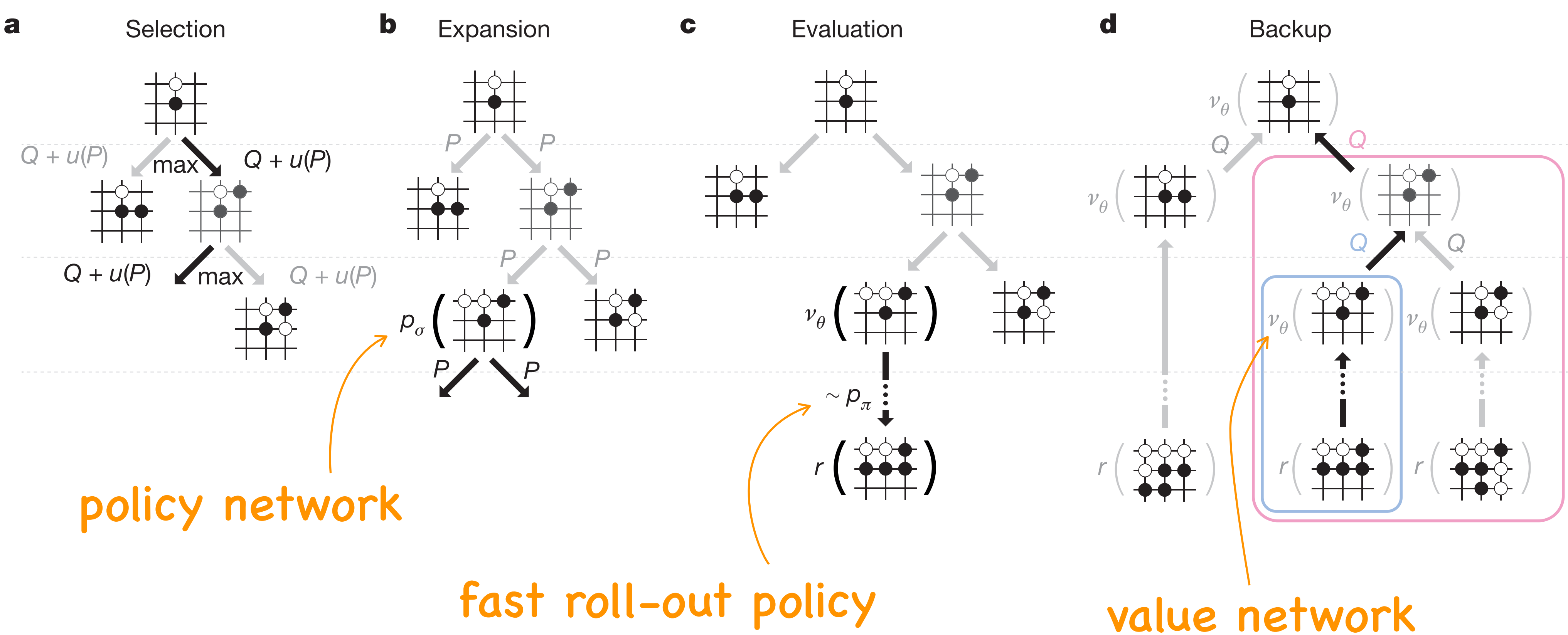


Improvements ?

- Put stones randomly in the neighborhood of a previous stone
- Put stones matching patterns prior knowledge
- Put stones optimizing a value function Silver et al. 07

AlphaGo

A combination of tree search, deep neural networks and reinforcement learning

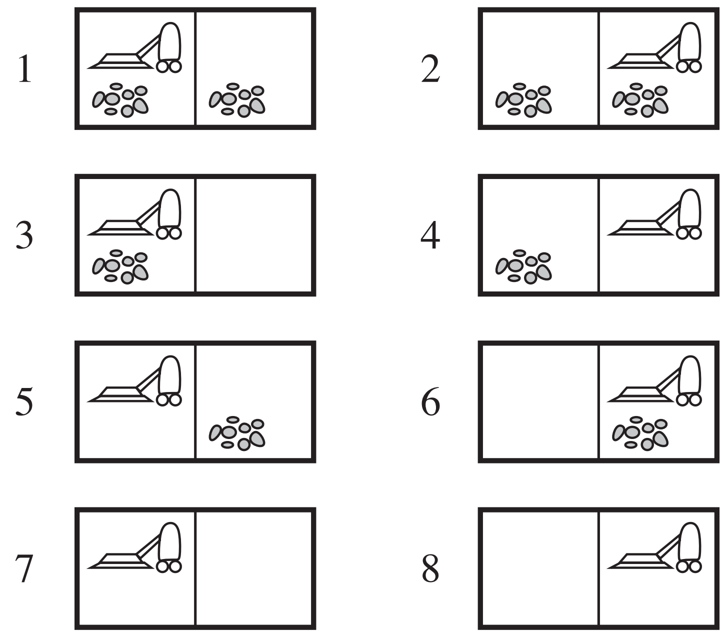


Different Environment Properties

Nondeterministic actions

In the **erratic vacuum world**, the *Suck* action works as follows:

- When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
- When applied to a clean square the action sometimes deposits dirt on the carpet.

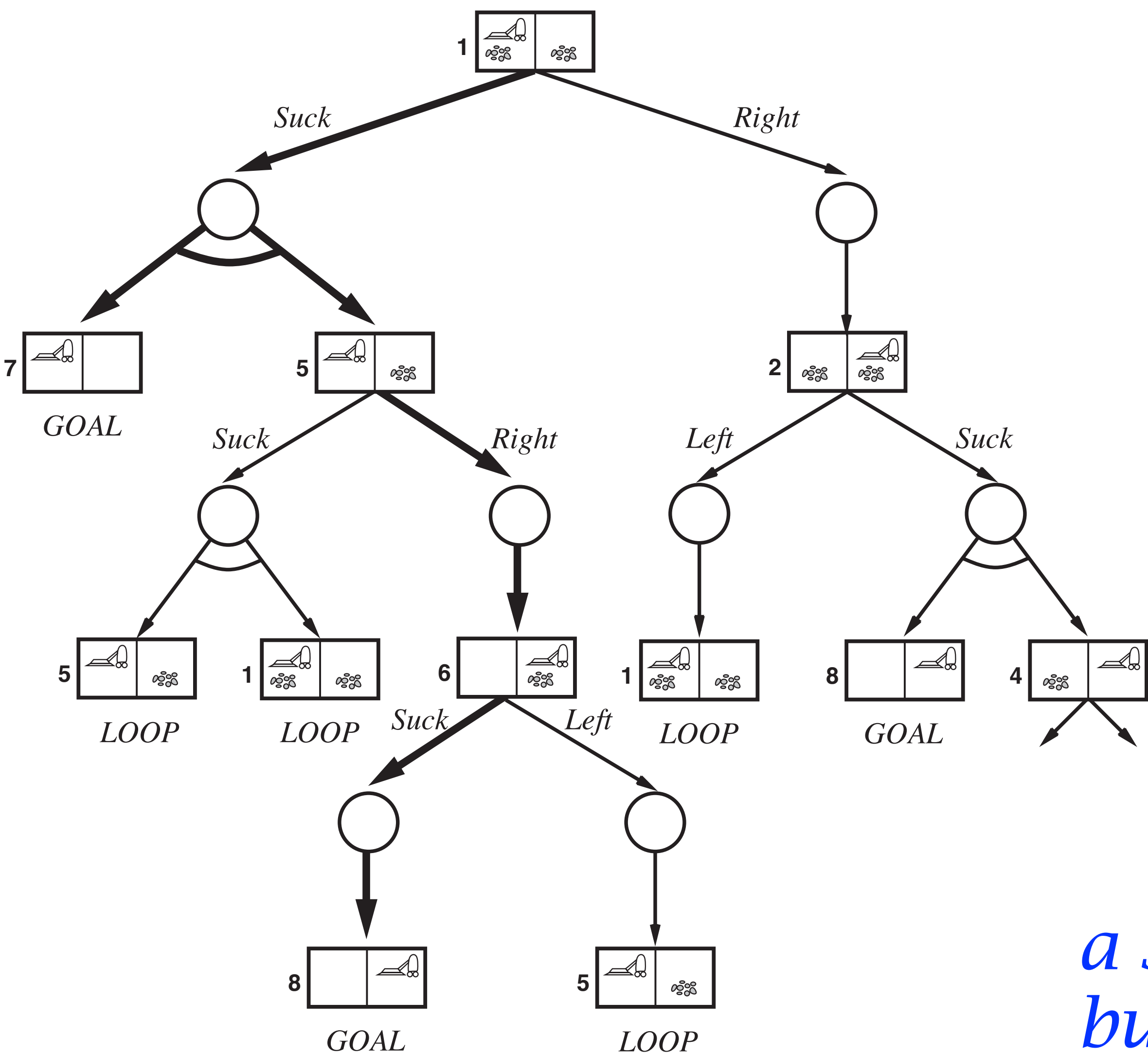


almost all real-world problems are nondeterministic

how do you solve this problem?

AND-OR tree search

OR node: different actions (as usual)
AND node: different transitions



a solution is not a path
but a tree

Depth-first AND-OR tree search

function AND-OR-GRAPH-SEARCH(*problem*) **returns** *a conditional plan, or failure*
  OR-SEARCH(*problem*.INITIAL-STATE, *problem*, [ ])

---

function OR-SEARCH(*state*, *problem*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.GOAL-TEST(*state*) **then return** the empty plan
  if *state* is on *path* **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    plan ← AND-SEARCH(RESULTS(*state*, *action*), *problem*, [*state* | *path*])
    if *plan* $\neq$ *failure* **then return** [*action* | *plan*]
  return *failure*

---

function AND-SEARCH(*states*, *problem*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do**
    $plan_i \leftarrow$ OR-SEARCH(s_i, *problem*, *path*)
    if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** $\ldots$ **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

Search with no observations

search in **belief (in agent's mind)**