

Lecture 5: Search 4 Bandits and MCTS

2024-2025 900 INCI 2

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



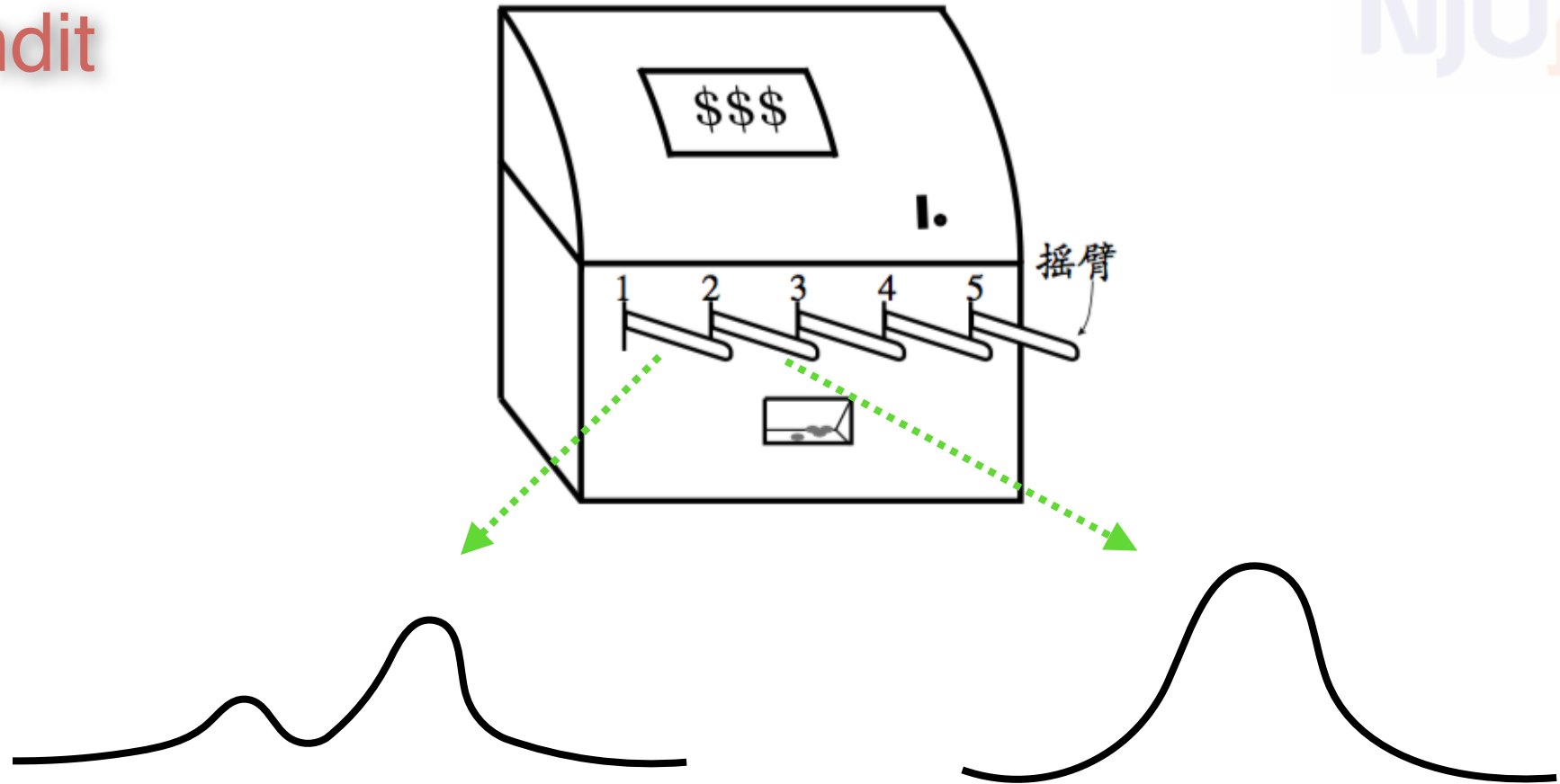
in C++

```
#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 trials and K arms, try each arm T/K times

problem?

Simplest strategies

Two simplest strategies

Exploration-only:

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

problem? waste on suboptimal arms

Simplest strategies

Two simplest strategies

Exploration-only:

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

problem? waste on suboptimal arms

Exploitation-only:

Simplest strategies

Two simplest strategies

Exploration-only:

for T trials 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

Simplest strategies



Two simplest strategies

Exploration-only:

for T trials 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?

Simplest strategies

Two simplest strategies

Exploration-only:

for T trials 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

Balance the exploration and exploitation:

with ϵ probability, try a random arm

with $1-\epsilon$ 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:

1: $r = 0$;

2: $\forall i = 1, 2, \dots, K : Q(i) = 0, \text{count}(i) = 0$;

3: **for** $t = 1, 2, \dots, T$ **do**

4: **if** $\text{rand}() < \epsilon$ **then**

5: $k =$ 从 $1, 2, \dots, K$ 中以均匀分布随机选取

6: **else**

7: $k = \arg \max_i Q(i)$

8: **end if**

9: $v = R(k)$;

10: $r = r + v$;

11: $Q(k) = \frac{Q(k) \times \text{count}(k) + v}{\text{count}(k) + 1}$;

12: $\text{count}(k) = \text{count}(k) + 1$;

13: **end for**

输出: 累积奖赏 r

Balance the exploration and exploitation:

Choose arm with probability

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

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:

```
1:  $r = 0$ ;  
2:  $\forall i = 1, 2, \dots, K : Q(i) = 0, \text{count}(i) = 0$ ;  
3: for  $t = 1, 2, \dots, T$  do  
4:    $k =$  从  $1, 2, \dots, K$  中根据式(16.4)随机选取  
5:    $v = R(k)$ ;  
6:    $r = r + v$ ;  
7:    $Q(k) = \frac{Q(k) \times \text{count}(k) + v}{\text{count}(k) + 1}$ ;  
8:    $\text{count}(k) = \text{count}(k) + 1$ ;  
9: end for
```

输出: 累积奖赏 r

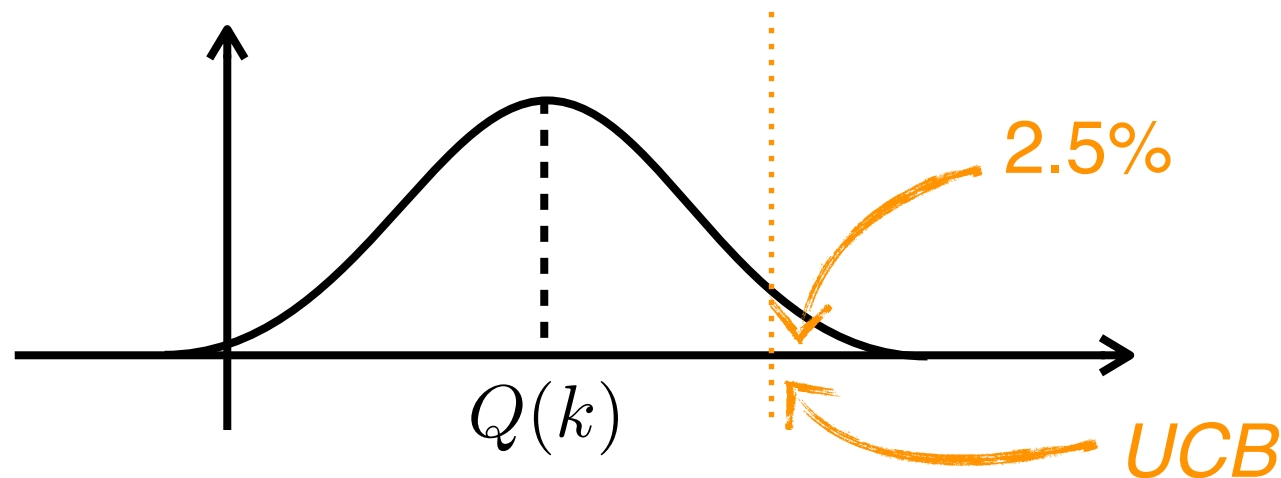
Upper-confidence bound

Balance the exploration and exploitation:

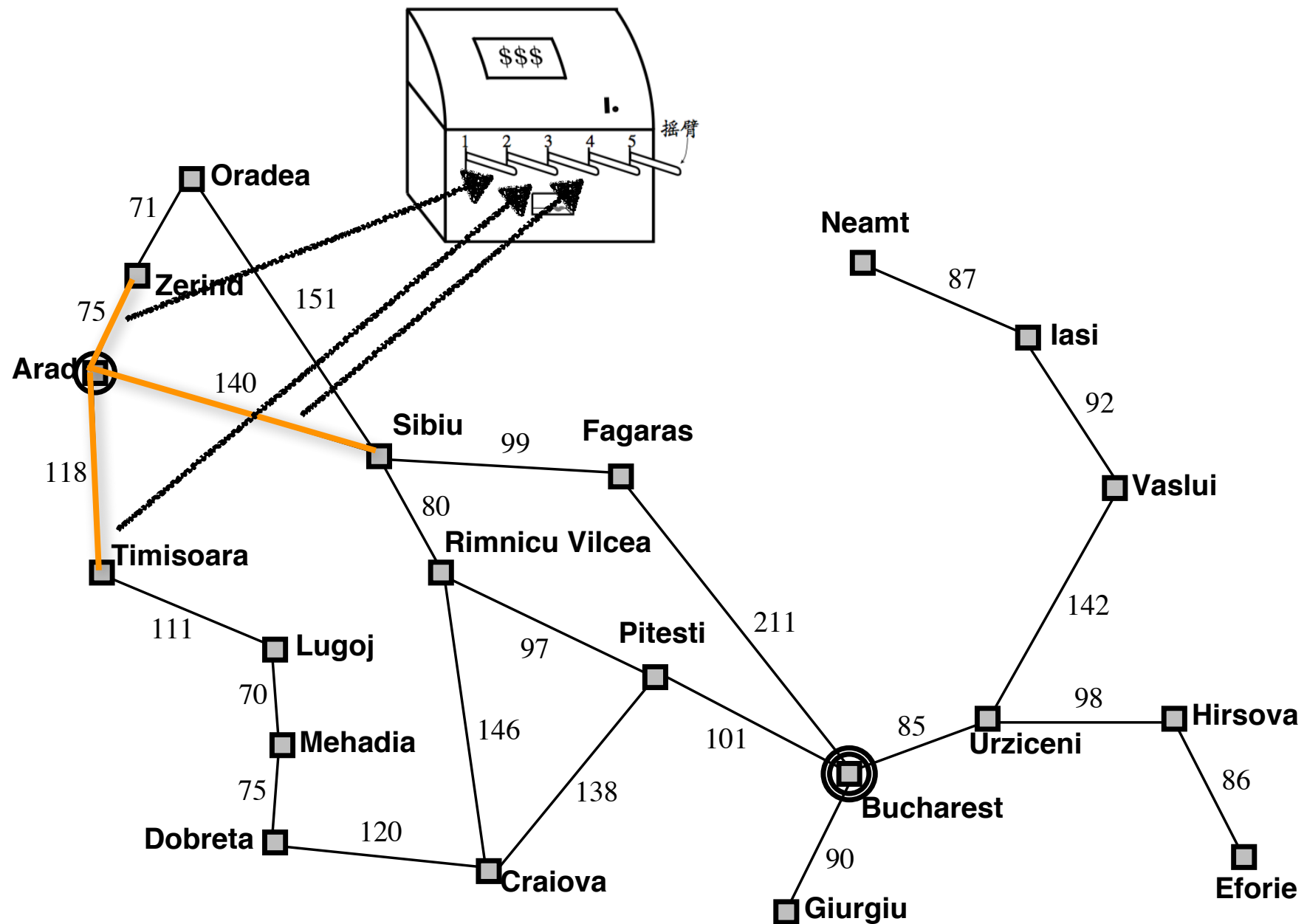
Choose arm with the largest value of

average reward + upper confidence bound

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

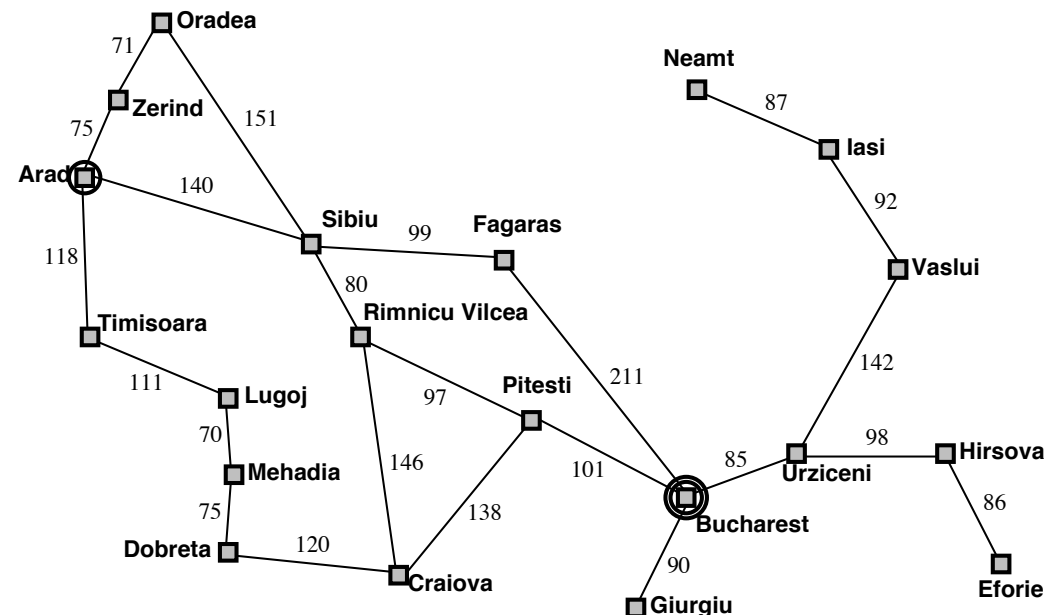


Use bandit to search

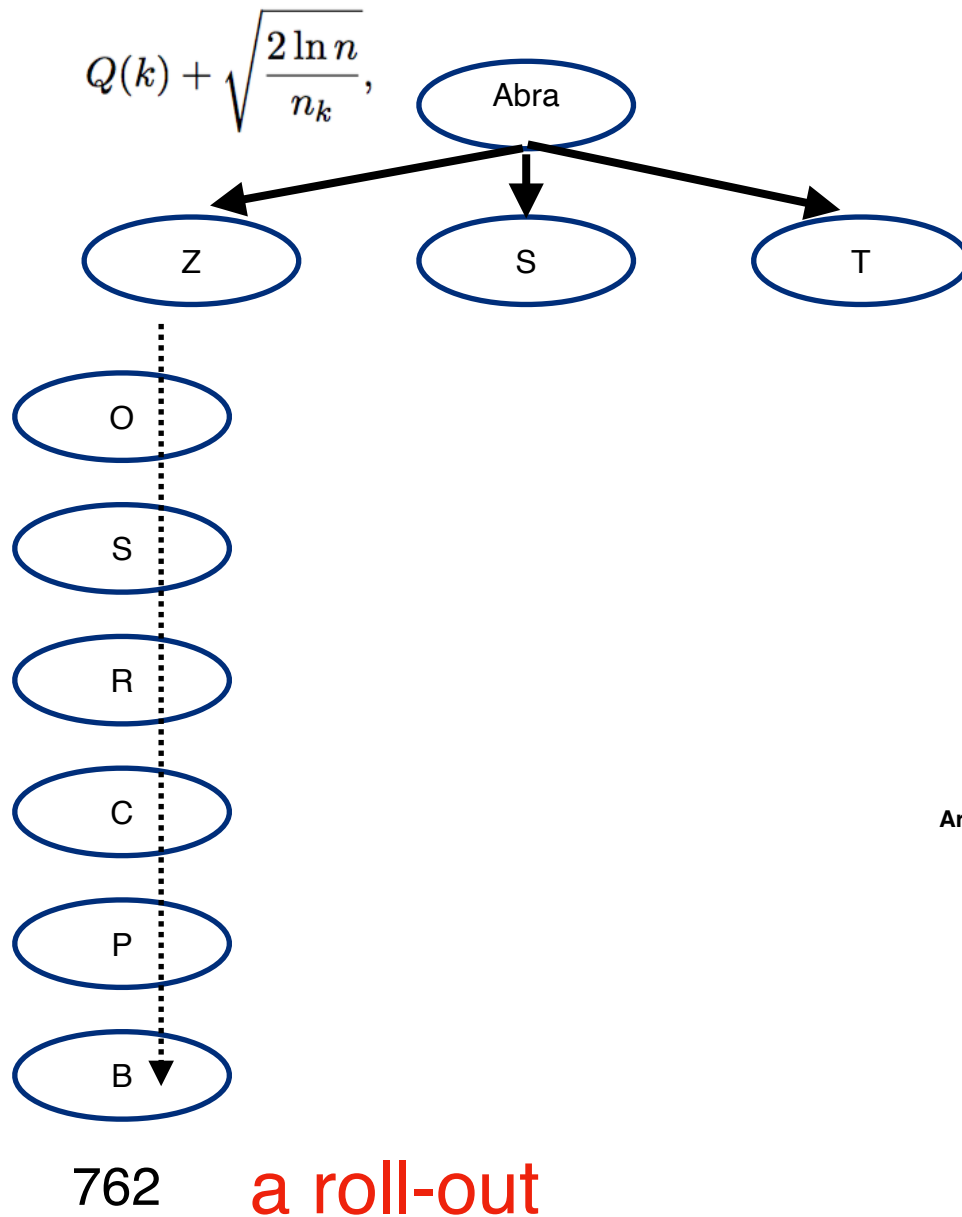




arm selection: UCB

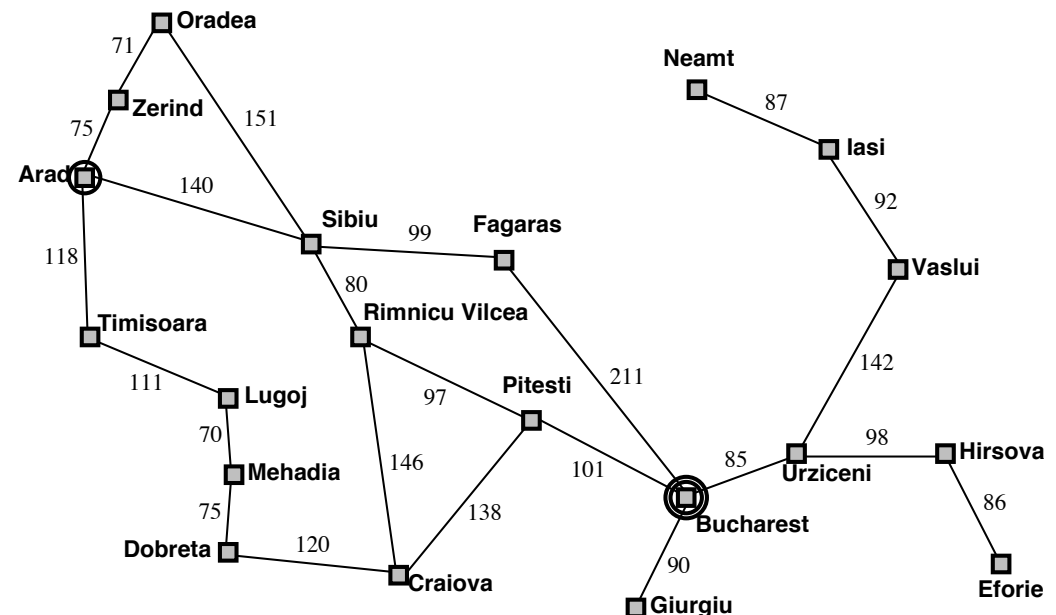


Use bandit to search



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

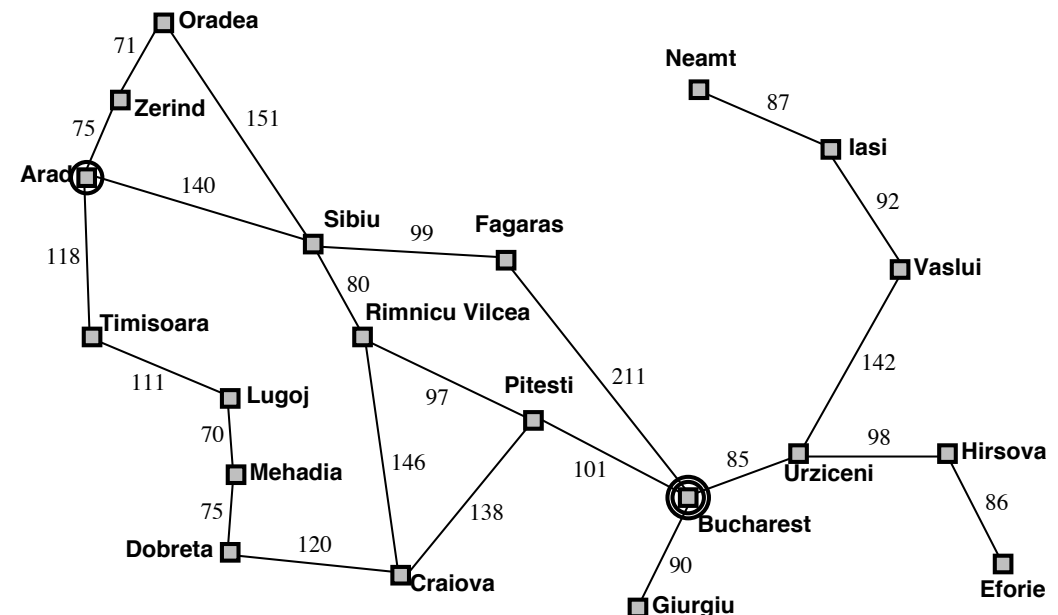
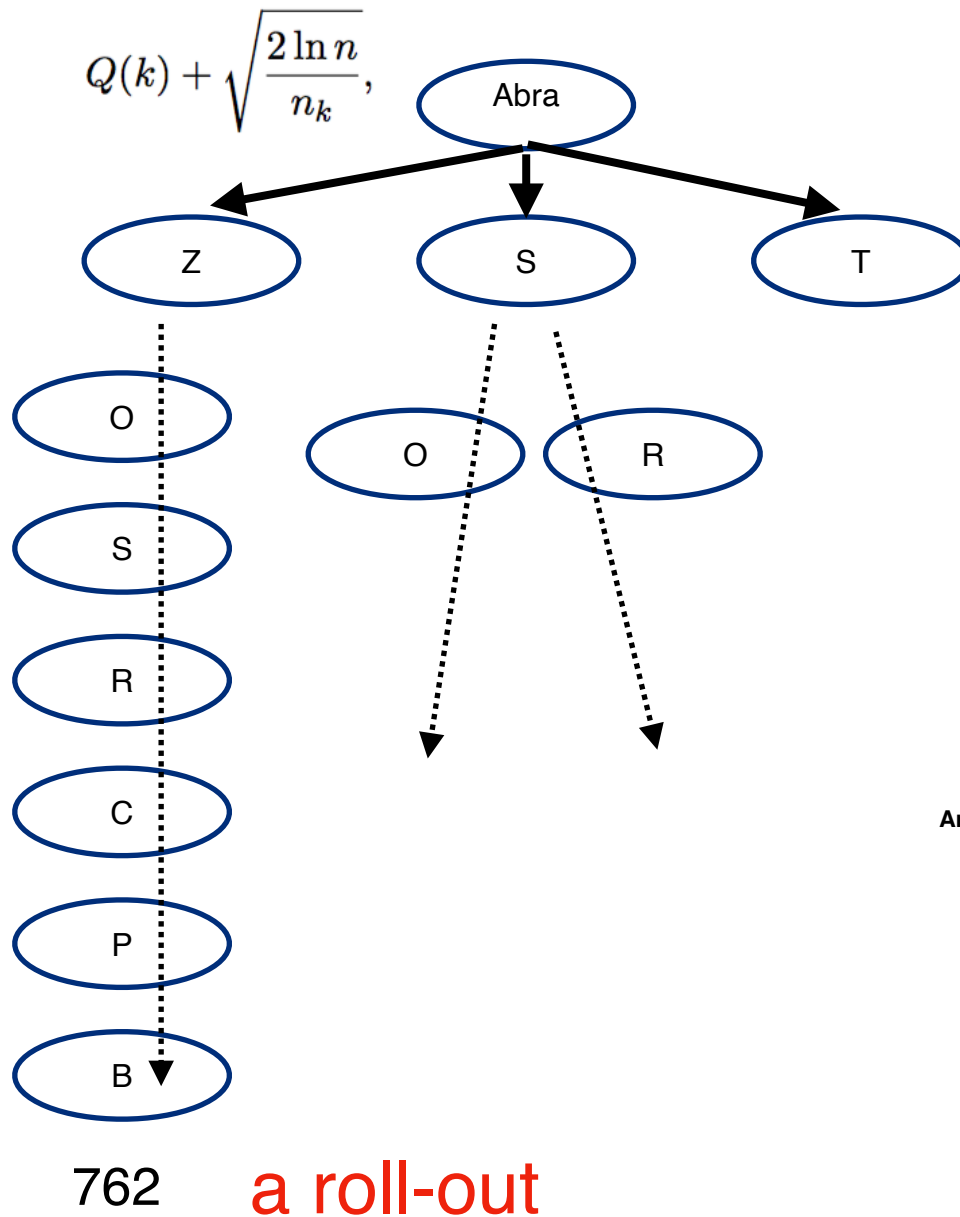
arm selection: UCB



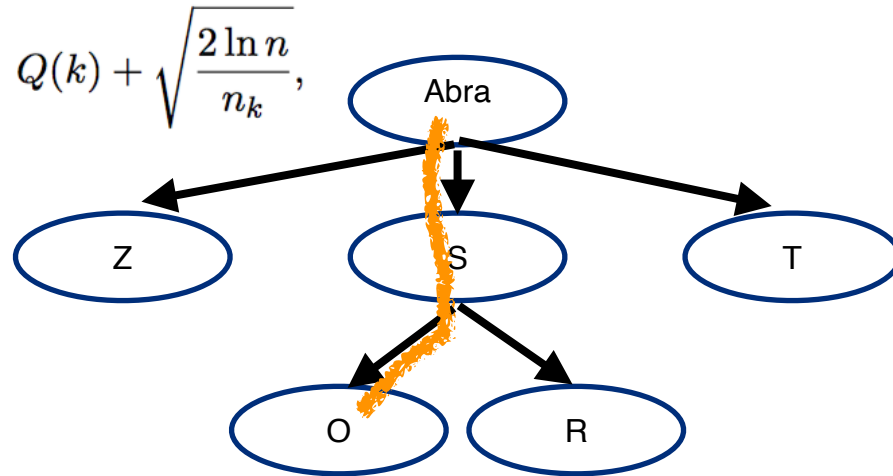
Use bandit to search

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

arm selection: UCB



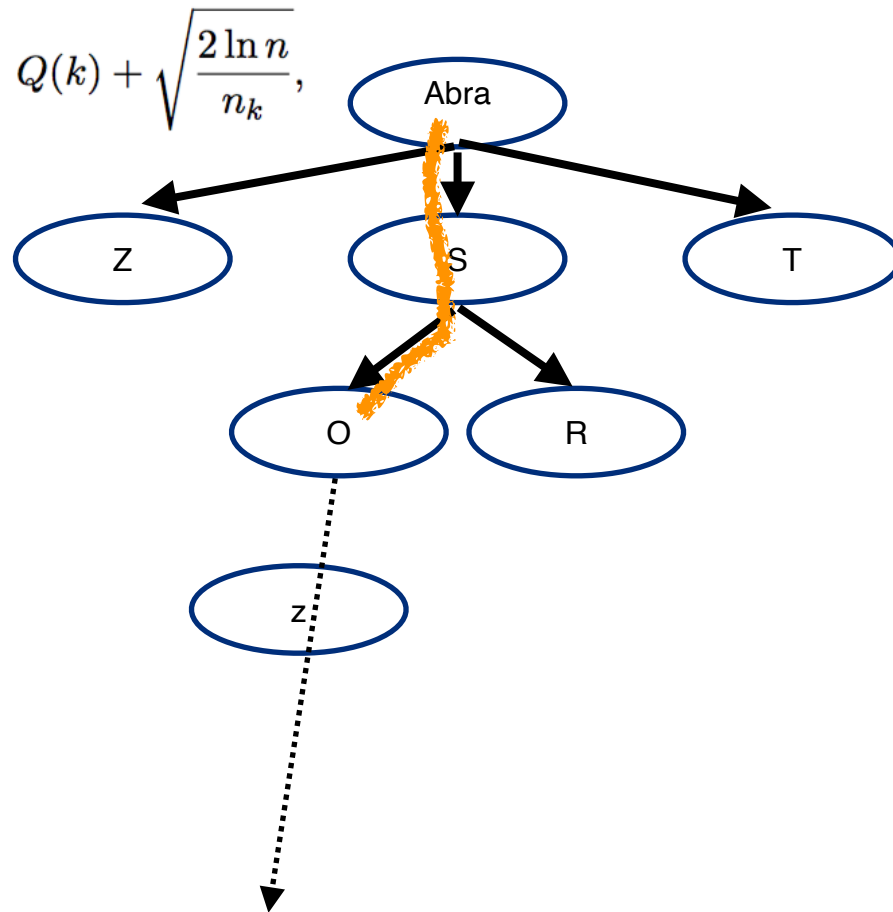
From bandit to tree



grow a tree

update the values along
the path

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
 - ▶ Add a node
 - ▶ Select next action bis
 - ▶ Compute instant reward
 - ▶ Update information in visited nodes
- ▶ Returned solution:
 - ▶ Path visited most often

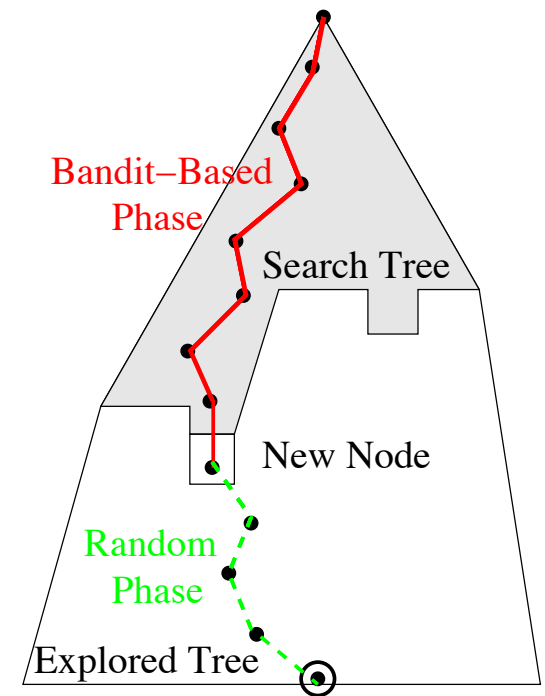
Bandit phase

Grow a leaf of the search tree

Random phase, roll-out

Evaluate

Propagate



Monte-Carlo Tree Search



```
public class TreeNode {
    static Random r = new Random();
    static int nActions = 5;
    static double epsilon = 1e-6;

    TreeNode[] children;
    double nVisits, totValue;

    public void selectAction() {
        List<TreeNode> visited = new LinkedList<TreeNode>();
        TreeNode cur = this;
        visited.add(this);
        while (!cur.isLeaf()) {
            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);
        }
    }
}
```

```
public void expand() {
    children = new TreeNode[nActions];
    for (int i=0; i<nActions; i++) {
        children[i] = new TreeNode();
    }
}
```

```
public void updateStats(double value) {
    nVisits++;
    totValue += value;
}
```

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;
}
```

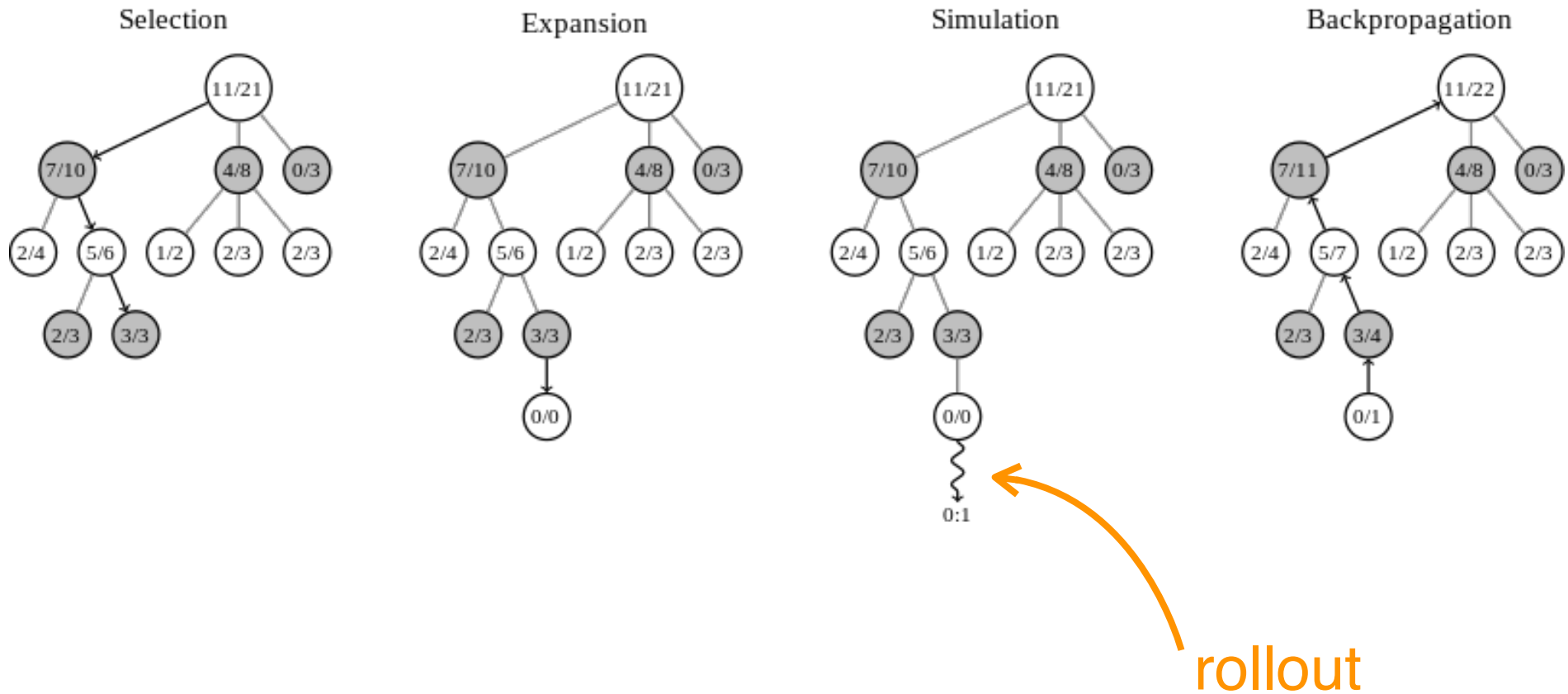
```
cur = cur.select();
visited.add(cur);
```

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

```
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);
}
}
```

Monte-Carlo Tree Search

Example:



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 $r = \text{Win}(\text{black})$
3. The outcome of the tree-walk is r

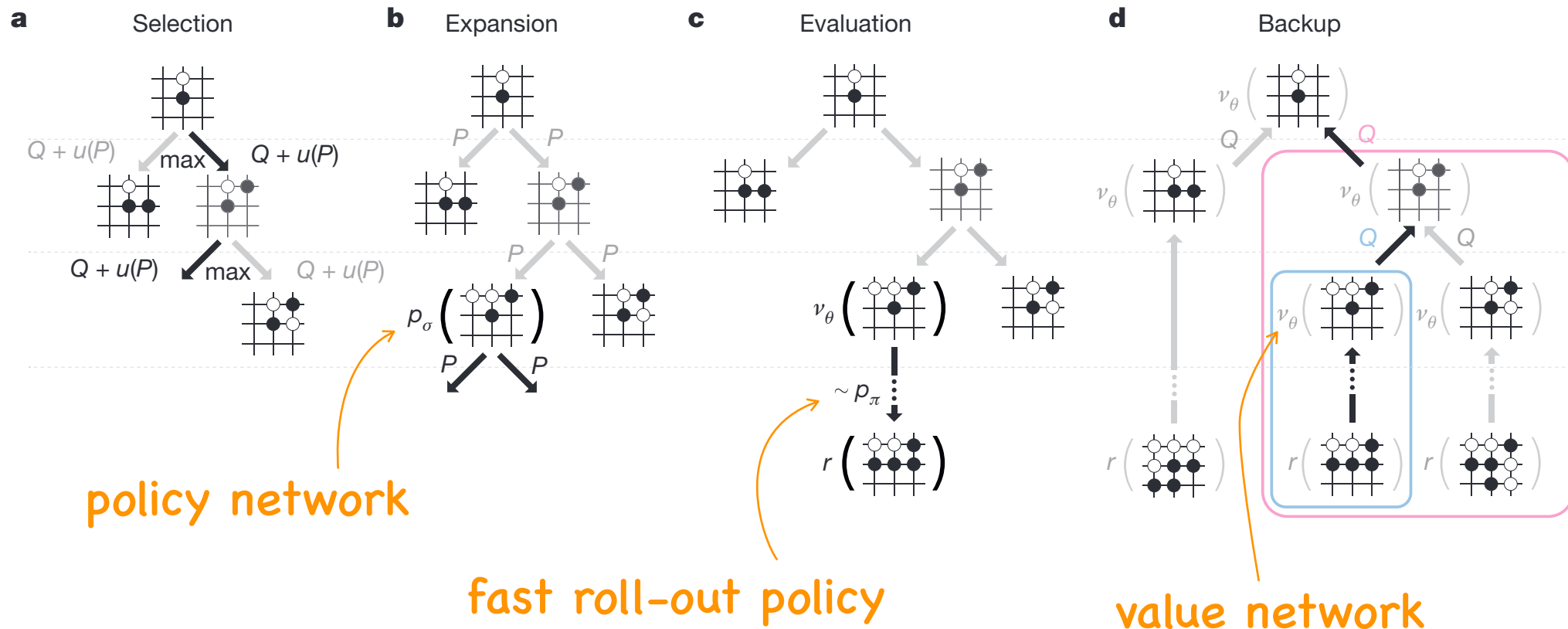


Improvements ?

- ▶ Put stones randomly in the neighborhood of a previous stone
- ▶ Put stones matching patterns
- ▶ Put stones optimizing a value function

Silver et al. 07

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

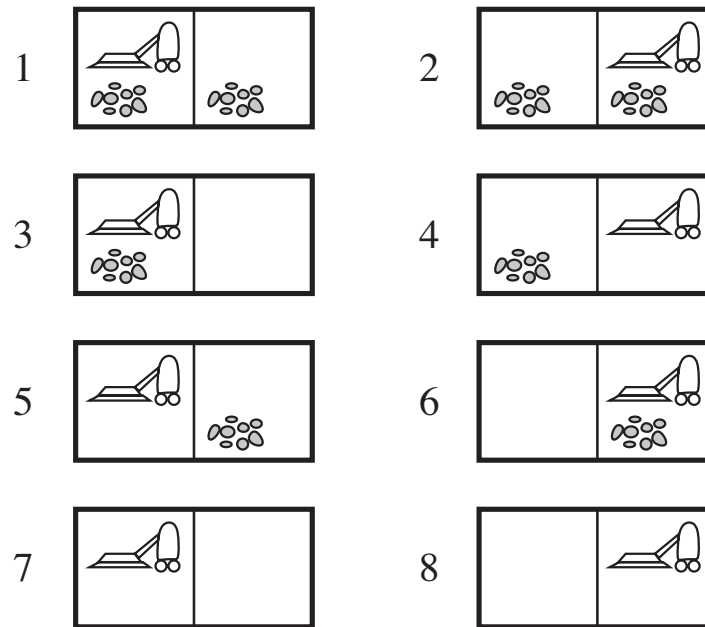


Different Environment Properties

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.

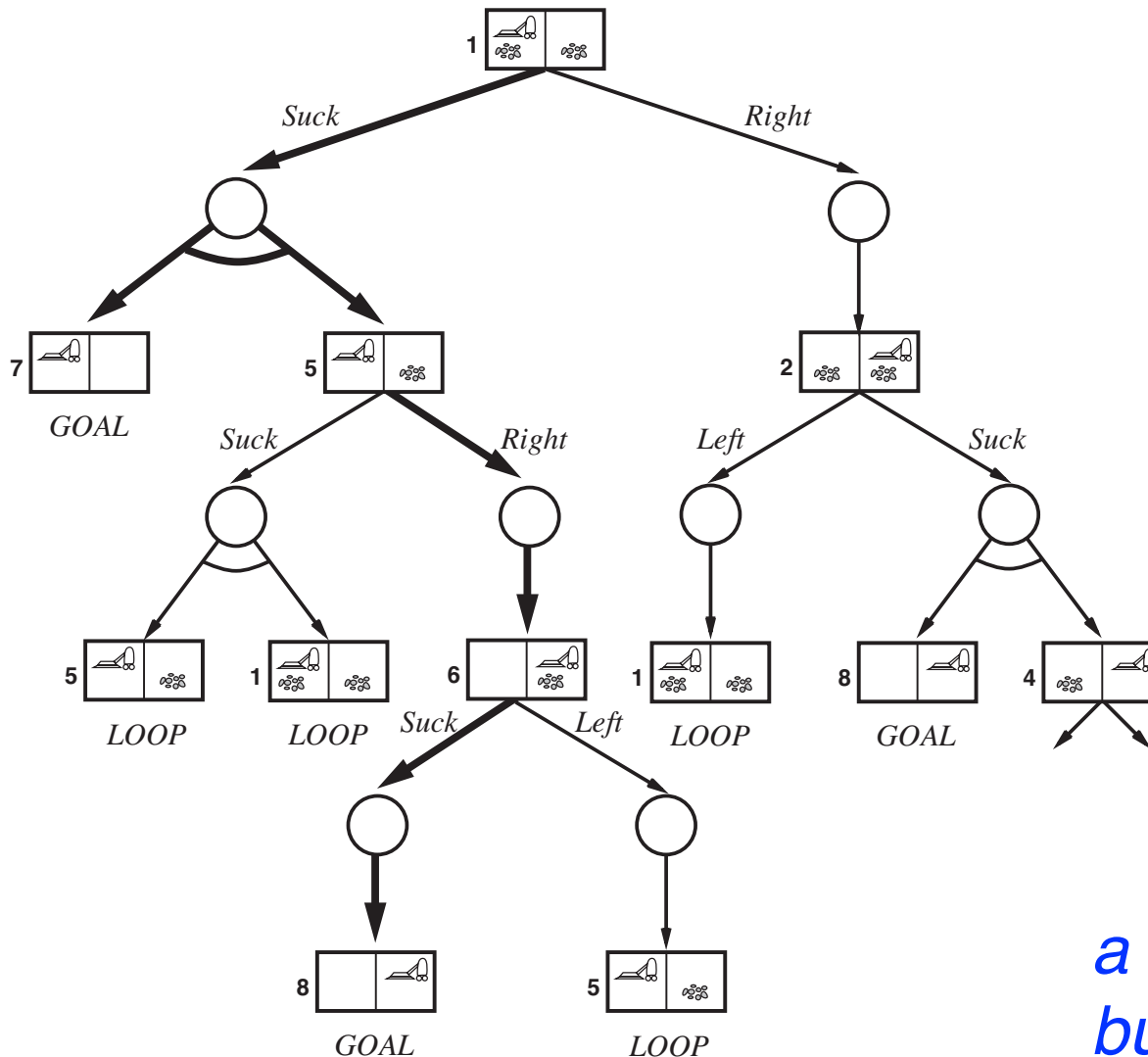


almost all real-world problems are nondeterministic
how do you solve this problem?

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 \leftarrow 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** ... **if** s_{n-1} **then** *plan* _{$n-1$} **else** *plan* _{n}]

Search with no observations

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

