MONTE-CARLO (TREE) SEARCH

Stefan Edelkamp

PUI - CTU

AI CENTER
FEE CTU

MONTE-CARLO TREE SEARCH

- General randomized heuristic search technique
- Less problem-specific knowledge to be added

Protagonists

- UCB/UCT, best in many games like Go, Amazons, GGP, etc. (Auer et al. 2002, Kocsis & Szepesvari, 2006, Coulom 2006)
- Nested-Monte-Carlo Tree Search (Cazenave, 2009)
- Nested Rollout with Policy Adaptation (Rosin, 2011)

Input: iteration width (exploit), nestedness level (explore)
Policy: (city-to-city) Mapping $N \times N \rightarrow IR$ to be learnt

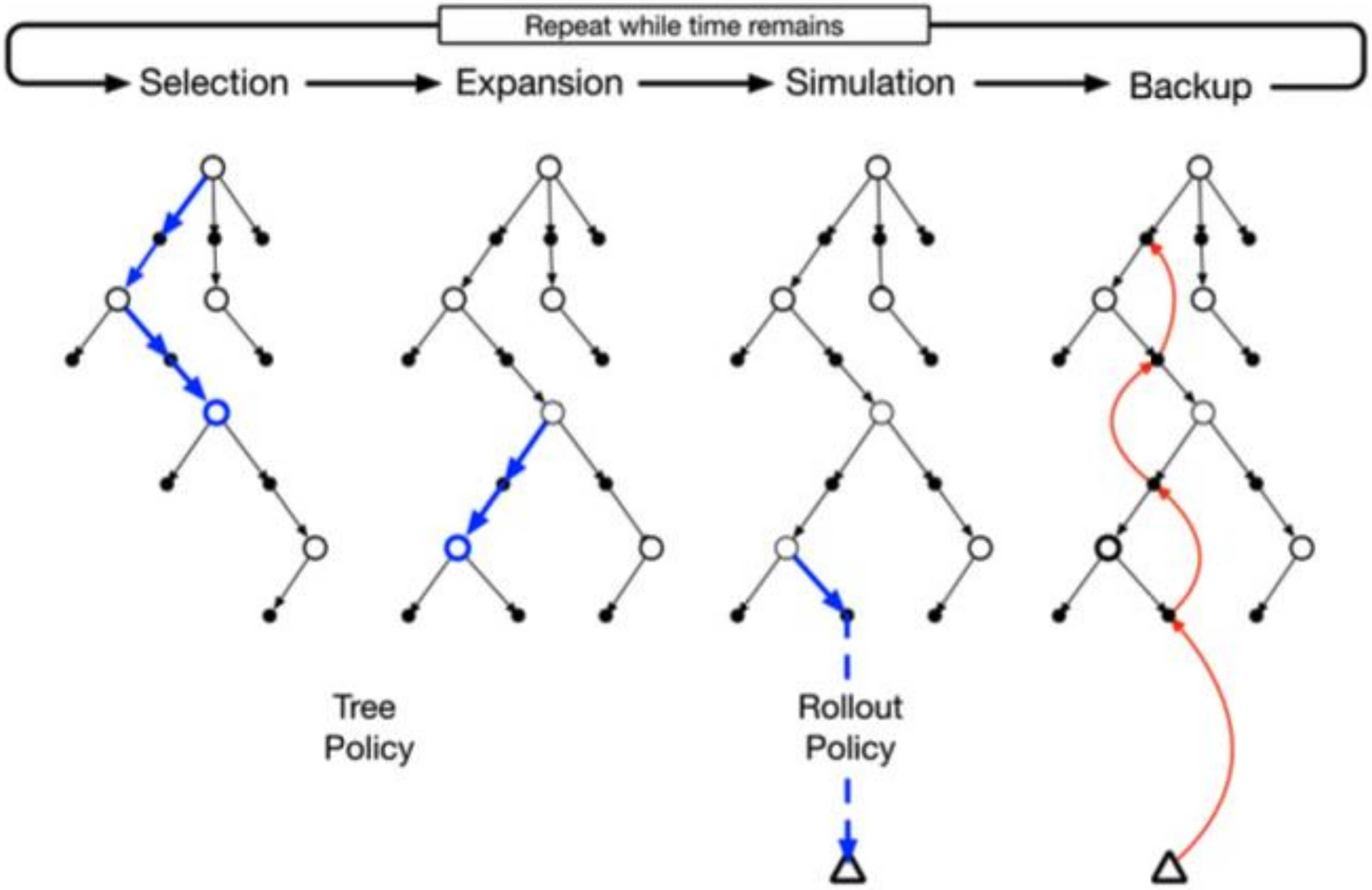
MONTE-CARLO SEARCH

Consider set of random walks from initial to terminal state

Set average reward

No further steering, no further knowledge

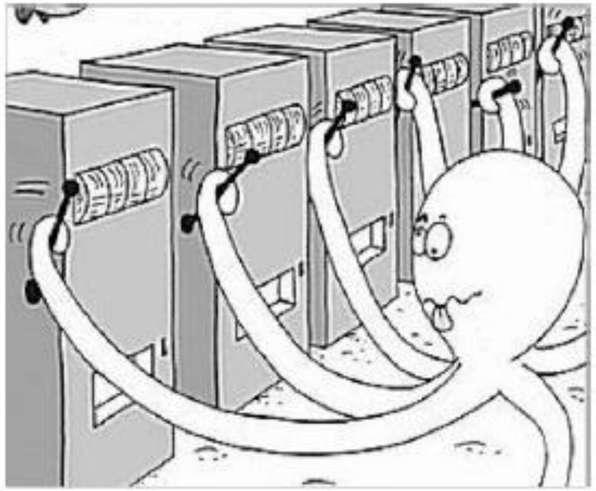
At end: Choose move with best evaluation



AI CENTER
FEE CTU

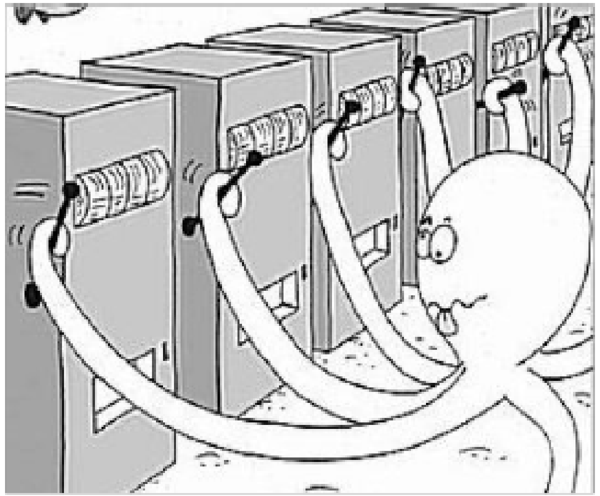
Multi-Arm Bandit Problem

- Statistical model of sequential experiments
  - Name comes from a traditional slot machine (one-armed bandit)
- Multiple actions $a_1, a_2, \ldots, a_n$
  - Each a_i provides a reward from an unknown (but stationary) probability distribution p_i
  - Objective: maximize expected utility of a sequence of actions
- Exploitation vs exploration dilemma:
  - ***Exploitation***: choose action that has given you high rewards in the past
  - ***Exploration***: choose action that you don't know much about, in hopes that it might produce a higher reward



UCB (Upper Confidence Bound) Algorithm

- Assume all rewards are between 0 and 1
  - If they aren't, normalize them
- For each action a_i , let
  - r_i = average reward you've gotten from a_i
  - t_i = number of times you've tried a_i
  - $t = \Sigma_i t_i$



loop

if there are one or more actions that you haven't tried

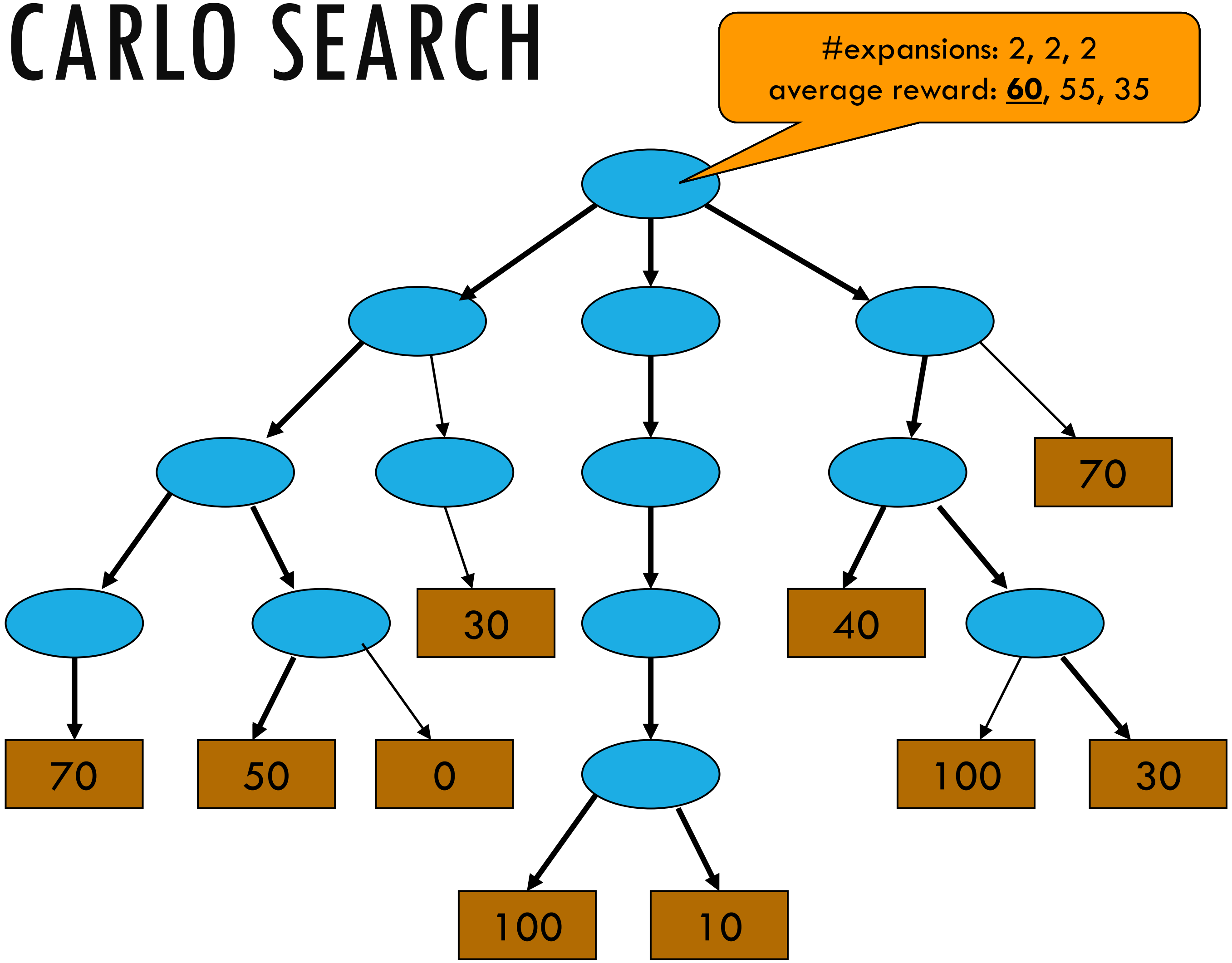
then choose an untried action a_i at random

else choose an action a_i that has the highest value of $r_i + \sqrt{2(\ln t)/t_i}$

perform a_i

update r_i, t_i, t

MONTE-CARLO SEARCH



MONTE-CARLO SEARCH

Advantage:

- Easy to implement
- Bettern than random
- Small memory requirement

Disadvantage:

- Repeated expansion of same states
  - Slowing down search
- No guidance
- Results potentially bad
- Information lost in further runs

MONTE-CARLO SEARCH WITH MEMORY

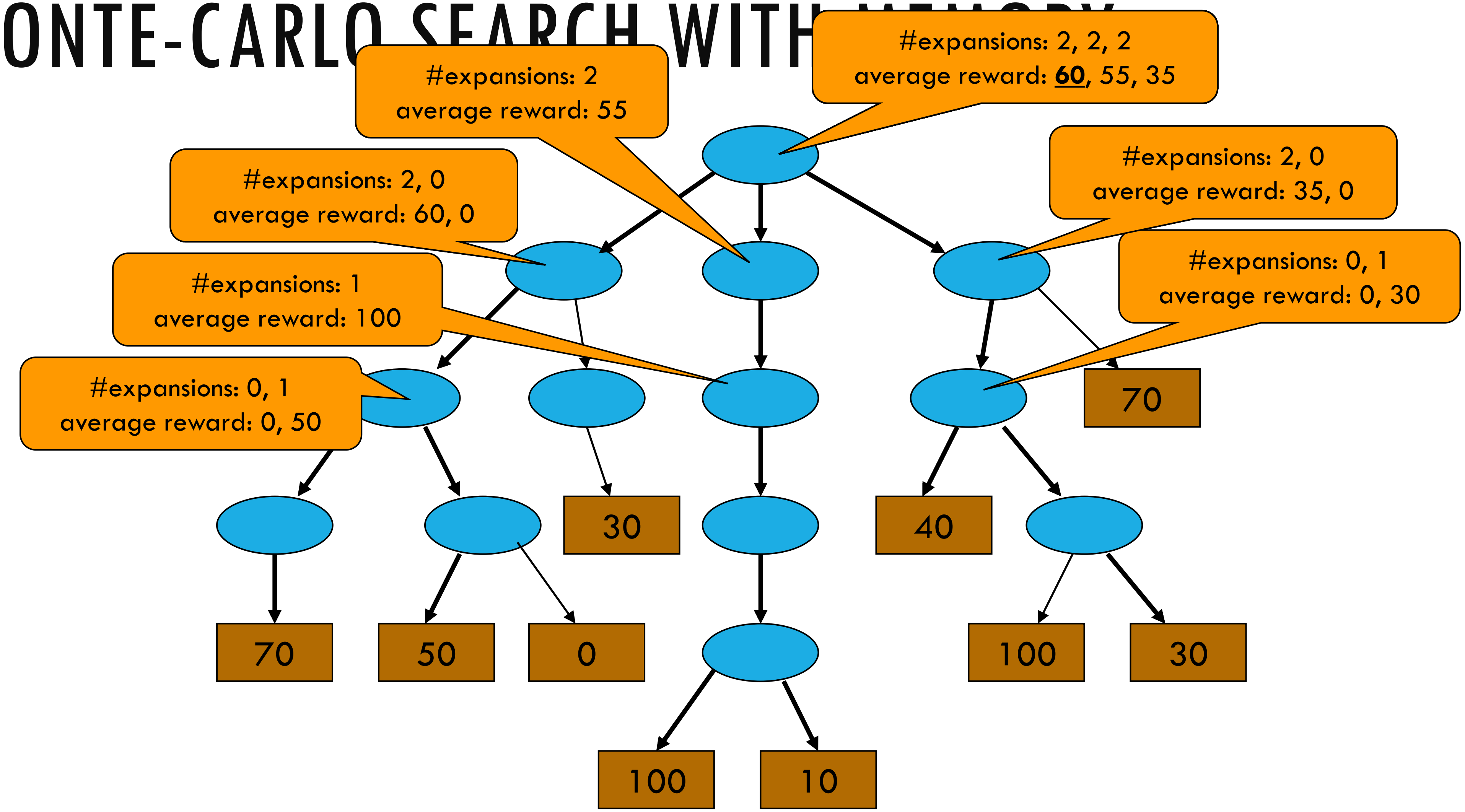
first + last problem resolved by using an explicit tree

- Encapulation of memory
- Instead forgetting about everything after Monte-Carlo run, insert node in tree
- Only on nod at a time because of memory requirement

Search in the tree remains random

- Expansions faster if successor stored in tree schneller
- Update for all nodes in the tree
  - Stored node used later on in the search

MONTE-CARLO SEARCH WITH MEMORY



MONTE-CARLO SEARCH WITH MEMORY

Advantages:

- Storage → less expansions
- Information of successors can be used in upcoming moves

Disadvantages:

- More momory requirements
- Unguided search
- Results often not good

UCT [KOCSIS & SZEPESVÁRI, 2006]

"Upper Confidence Bounds applied to Trees"

Additional Information

- In the Tree
  - If ≥ 1 unexpanded move, chose a random one
  - Choose successor, maximizing UCT value

$$Q(s,m)+C\sqrt{\frac{\ln(N(s))}{N(s,m)}}$$

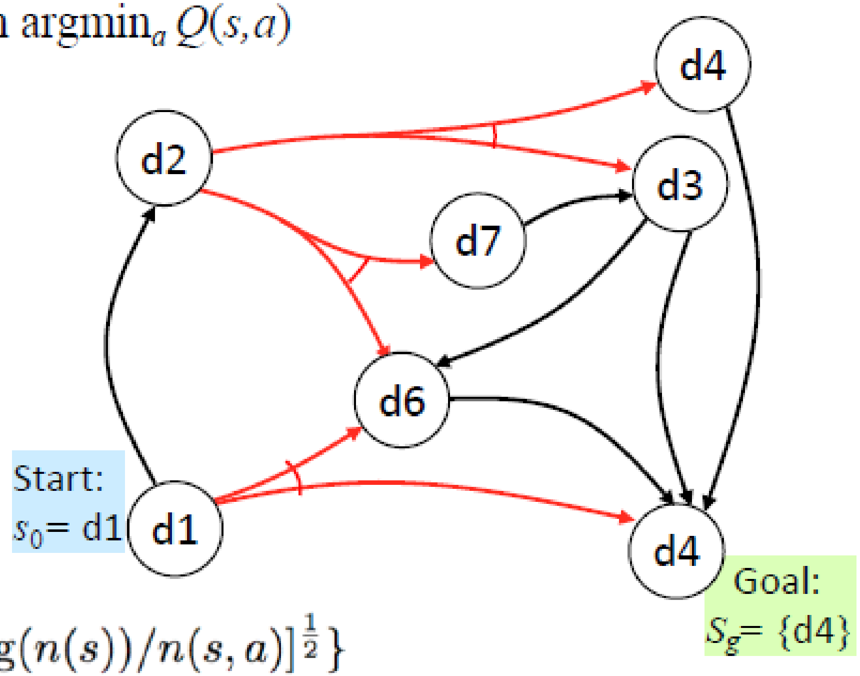
  - Q(s, m) average reward of move m in state s
  - C: constant
  - N(s): number visits of state s
  - N(s, m): number visits of state s with chosen move m
  - If leaf found
  - Normal Monte-Carlo Suche d
  - Backpropagate terminal evaluation up the tree.

AI CENTER
FEE CTU

UCT Algorithm

- Recursive UCB computation to compute $Q(s,a)$
- Anytime algorithm: call repeatedly until time runs out
  - Then choose action $\text{argmin}_a\, Q(s,a)$

$\mathsf{UCT}(s, h)$
  if $s \in S_g$ then return 0
  if $h = 0$ then return $V_0(s)$
  if $s \notin \mathit{Envelope}$ then do
    add s to $\mathit{Envelope}$
    $n(s) \leftarrow 0$
    for all $a \in \text{Applicable}(s)$ do
      $Q(s,a) \leftarrow 0;\ \ n(s,a) \leftarrow 0$
  $\mathit{Untried} \leftarrow \{a \in \text{Applicable}(s) \mid n(s,a) = 0\}$
  if $\mathit{Untried} \neq \varnothing$ then $\tilde{a} \leftarrow \mathsf{Choose}(\mathit{Untried})$
  else $\tilde{a} \leftarrow \text{argmin}_{a \in \text{Applicable}(s)}\{Q(s,a) - C \times [\log(n(s))/n(s,a)]^{\frac{1}{2}}\}$
  $s' \leftarrow \mathsf{Sample}(\Sigma, s, \tilde{a})$
  $\mathit{cost\text{-}rollout} \leftarrow \text{cost}(s,\tilde{a}) + \mathsf{UCT}(s', h-1)$
  $Q(s,\tilde{a}) \leftarrow [n(s,\tilde{a}) \times Q(s,\tilde{a}) + \mathit{cost\text{-}rollout}]/(1 + n(s,\tilde{a}))$
  $n(s) \leftarrow n(s) + 1$
  $n(s,\tilde{a}) \leftarrow n(s,\tilde{a}) + 1$
  return $\mathit{cost\text{-}rollout}$

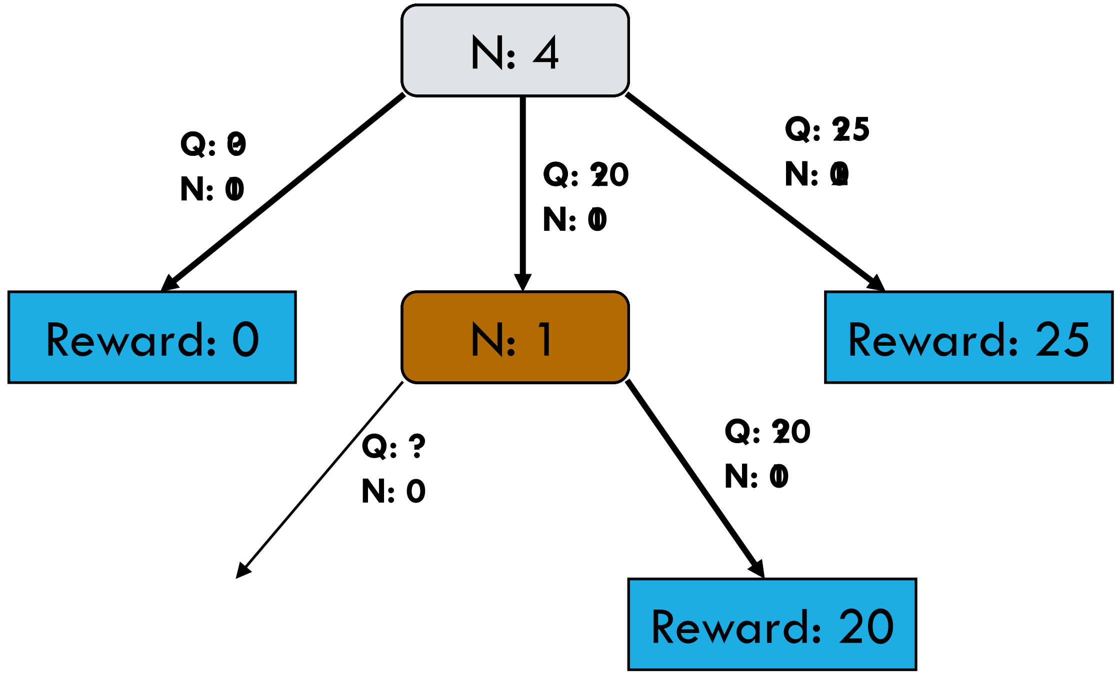


UCT in Two-Player Games

- Generate Monte Carlo rollouts using a modified version of UCT
- Main differences:
  - Instead of choosing actions that minimize accumulated cost, choose actions that maximize payoff at the end of the game
  - UCT for player 1 recursively calls UCT for player 2
    - Choose opponent's action
  - UCT for player 2 recursively calls UCT for player 1

- This produced the first computer programs to play go well
  - ≈ 2008–2012
- Monte Carlo rollout techniques similar to UCT were used to train AlphaGo

UCT



$$Q(s,m)+C\sqrt{\frac{\ln(N(s))}{N(s,m)}}$$

for C = 10:
Move 1: 12
Move 2: 32
Move 3: 35

for C = 100:
Move 1: 108
Move 2: 138
Move 3: 130

for C = 1,000:
Move 1: 1078
Move 2: 1098
Move 3: 1073

IMPROVEMENTS UCT

[Finnsson & Björnsson, 2010]

- Move-Average Sampling Technique (MAST)
- Tree-Only MAST (TO-MAST)
- Predicate-Average Sampling Technique (PAST)
- Features-to-Action Sampling Technique (FAST)
- Rapid Action Value Estimation (RAVE)

MAST

Idea:

- In every UCT-run, refine knowledge about all the moves
- Use knowledge to improve knowledge to steer search outside the tree

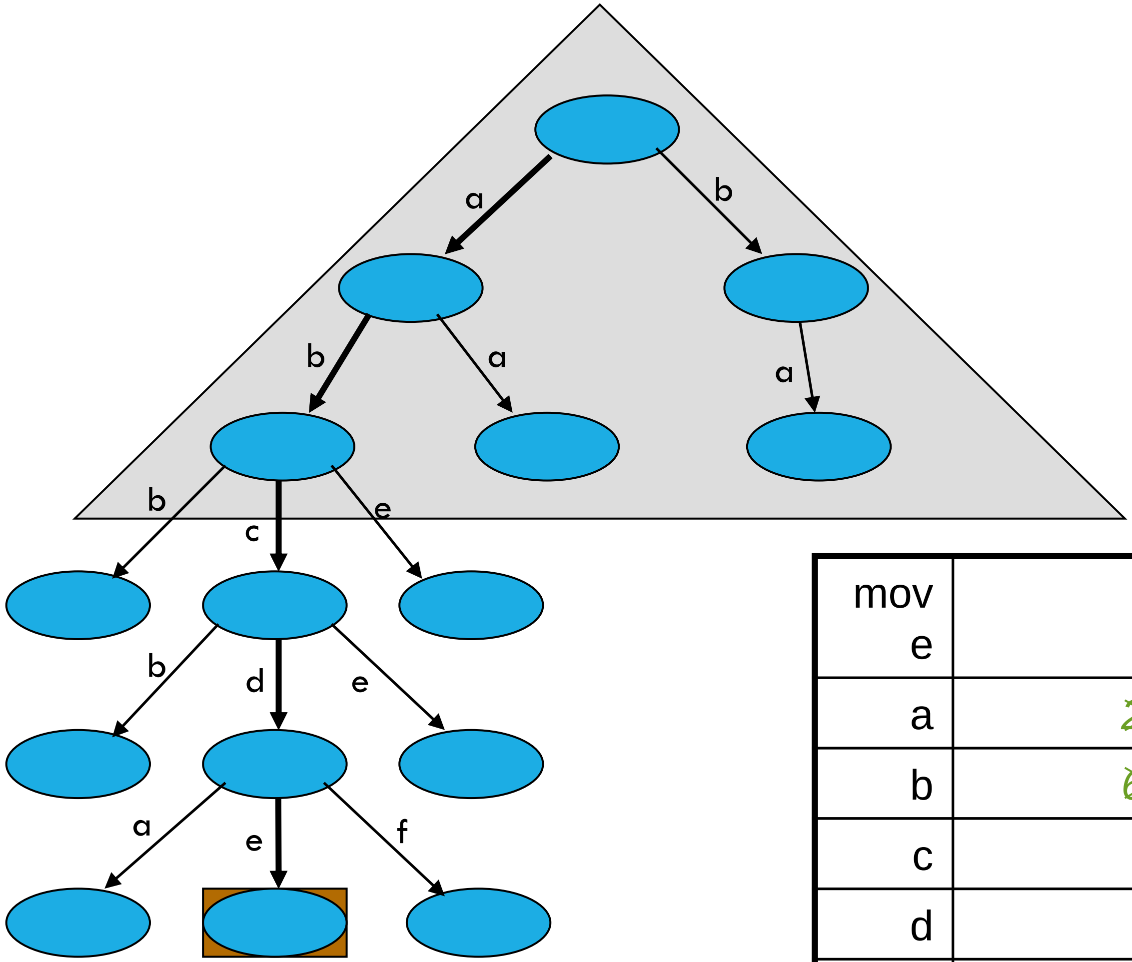
Maintain average reward for every move

- Independent of state

After UCT-run: update average reward of all the moves

- Moves, chosen frequently (independent of state) receive better score
- Hope: Moves are often good if available
  - e..g. placement of a stone in the corner of Reversi
  - e.g. Taking an opponent stone before the own base in Breakthrough

MOVE-AVERAGE SAMPLING TECHNIQUE



move	average	#Visits
a	~~25~~ **41,67**	~~2~~ **3**
b	~~60~~ **63,75**	~~3~~ **4**
c	~~15~~ **35**	~~2~~ **3**
d	~~85~~ **83**	~~4~~ **5**
e	~~10~~ **20,83**	~~5~~ **6**
f	70	1

MOVE-AVERAGE SAMPLING TECHNIQUE

Within Monte-Carlo run (outside UCT tree)

Choose move according to Gibbs sampling

$$P(m) = \frac{e^{Q_h(m)/\tau}}{\sum_{b=1}^{n} e^{Q_h(b)/\tau}}$$

with

m: chosen move

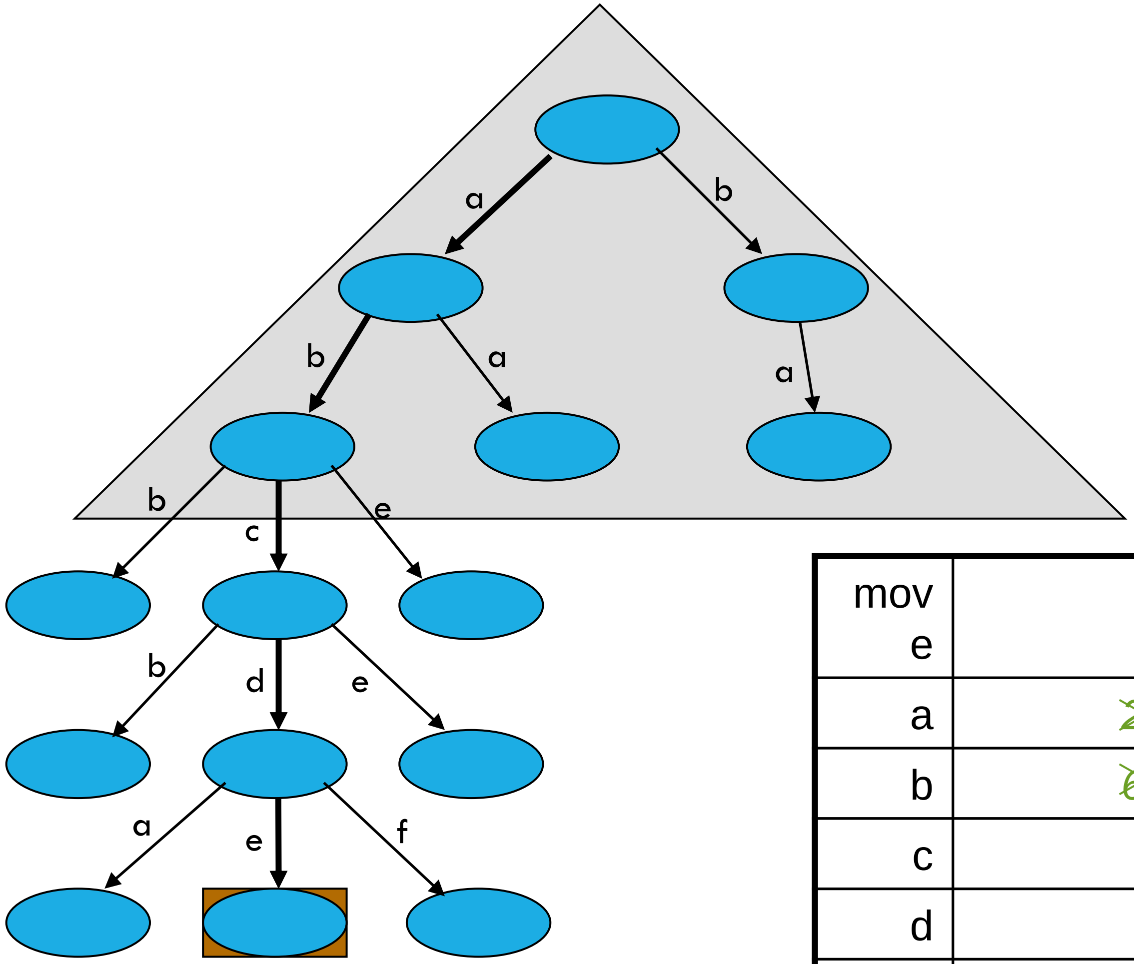
$Q_h(m)$: average reward of m

r: constant to tune (large value: closer to uniform disttribution)

TREE-ONLY MAST

(initial) Monte-Carlo runs random
may influence reward of actions negatively
Idea: use only results in UCT-tree
precise: apply UCT (with MAST) as before
but: update only moves, chosen within UCT tree
(ignore Monte-Carlo run)
In Monte-Carlo run: choose move according to MAST distribution

TO-MOVE-AVERAGE SAMPLING TECHNIQUE



move	average	#visits
a	~~25~~ **41,67**	~~2~~ **3**
b	~~60~~ **63,75**	~~3~~ **4**
c	15	2
d	85	4
e	10	5
f	70	1

RAPID ACTION VALUE ESTIMATION

First used in UCT Go AI [Gelly & Silver, 2007]

- known as all-moves-as-first geuristik

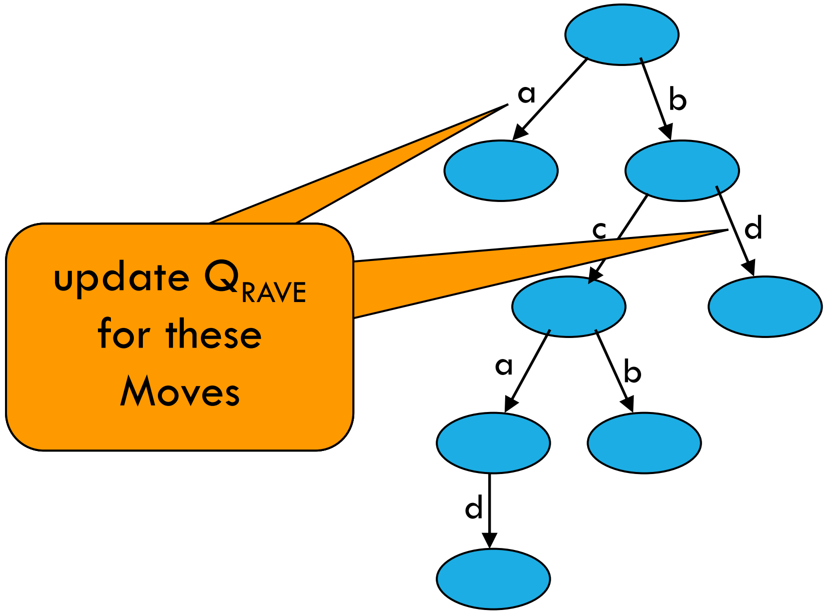
Accelerates learning withing UCT tree

Use later chosen moves, to derive more move samplings for the same but not chosen moves

RAPID ACTION VALUE ESTIMATION

Within UCT Tree

- update Q(s, m), if move m chosen in state s (as before)
- update Q_{RAVE}(s, m‘) for state s for non-chosen move m‘, if m‘ chosen later on in UCT tree



RAPID ACTION VALUE ESTIMATION

Issue: drift of average

- Good if less samples are available
- Choose only in case of higher variance in Q(s, m)
- Later on, Q(s, m) is more reliable
- Then ignore $Q_{RAVE}(s, m)$

➔

- Store RAVE values on top
- Choose weighted combination: ß(s) * $Q_{RAVE}(s, m)$ + (1 - ß(s)) * Q(s, m) in UCT choice
- with
  - $$\beta(s) = \sqrt{\frac{k}{3N(s)+k}}$$
  - k: parameter (deciding after how many samples weighted is equal)
  - N(s): number of visitis of s

COMPARISON

[Finnsson & Björnsson, 2010] Comparions UCT

Game	MAST win %	TO-MAST win %	PAST win %	RAVE win %	FAST win %
Breakthrough	90.00 (± 3.40)	85.33 (± 4.01)	85.00 (± 4.05)	63.33 (± 5.46)	81.67 (± 4.39)
Checkers	56.00 (± 5.37)	82.17 (± 4.15)	57.50 (± 5.36)	82.00 (± 4.08)	50.33 (± 5.36)
Othello	60.83 (± 5.46)	50.17 (± 5.56)	67.50 (± 5.24)	70.17 (± 5.11)	70.83 (± 5.10)
Skirmish	41.33 (± 5.18)	48.00 (± 5.29)	42.33 (± 5.16)	46.33 (± 5.30)	96.33 (± 1.86)

Comparison MAST

Game	TO-MAST win %	PAST win %	RAVE win %	FAST win %
Breakthrough	52.33 (± 5.66)	45.67 (± 5.65)	20.33 (± 4.56)	39.67 (± 5.55)
Checkers	82.00 (± 4.18)	55.83 (± 5.35)	78.17 (± 4.36)	46.17 (± 5.33)
Othello	40.67 (± 5.47)	49.17 (± 5.60)	58.17 (± 5.49)	56.83 (± 5.55)
Skirmish	56.00 (± 5.31)	43.33 (± 5.26)	59.83 (± 5.15)	97.00 (± 1.70)

RESULTS

[Finnsson & Björnsson, 2010]

UCT vs RAVE+MAST (RM) vs RAVE+FAST (RF)

Game	RM win %	RF win %
Breakthrough	89.00 (± 4.35)	76.50 (± 5.89)
Checkers	84.50 (± 4.78)	77.00 (± 5.37)
Othello	79.75 (± 5.52)	81.00 (± 5.32)
Skirmish	45.00 (± 6.55)	96.00 (± 2.34)

MAST vs RAVE+MAST (RM) vs. RAVE+FAST (RF)

Game	RM win %	RF win %
Breakthrough	50.50 (± 6.95)	38.50 (± 6.76)
Checkers	83.50 (± 4.87)	74.00 (± 5.81)
Othello	73.75 (± 6.01)	66.00 (± 6.43)
Skirmish	53.00 (± 6.47)	97.00 (± 2.04)

NESTED ROLLOUT POLICY ADAPTATION

Rosin 2011 IJCAI – Best Paper, Morpion Solitaire with new Record

MCS tree based on Complete Rollout and Recursive Search

Not really MCTS, No Search Tree.

In Each Level a Policy is Maintained, Updated and Refreshed

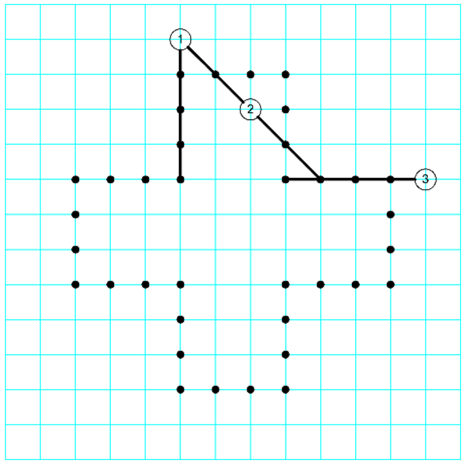
Updating Policy based on better Solutions Coming in from below

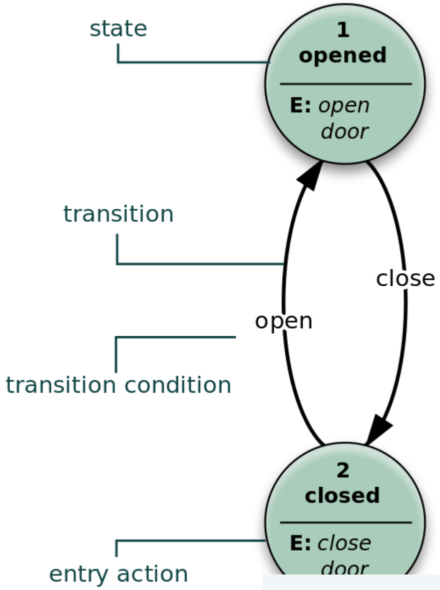
Policy in Turn Influences the Rollouts

Parameters: Level of Recursion, and Iteration Width

Effective for TSPTW and many other Approaches

Refinements: Beam / Diversity / Generalization



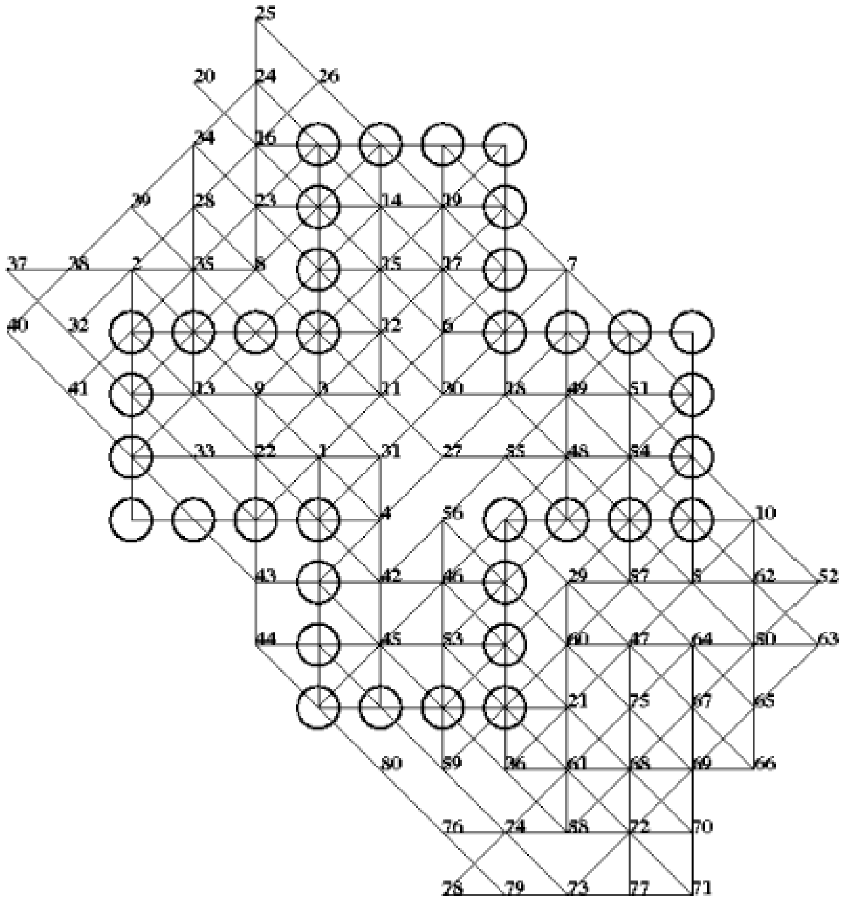


AI CENTER
FEE CTU

NEW RECORD

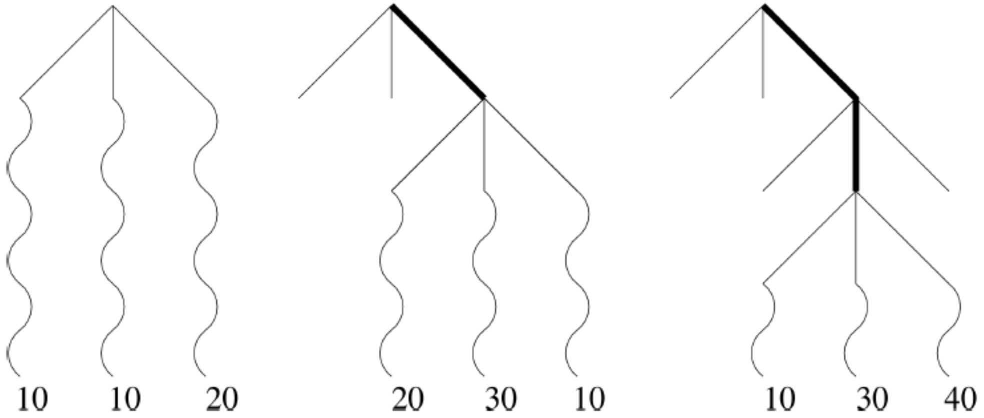
Morpion Solitaire

- 80 moves :



NESTED MCS

Nested Monte-Carlo Search



Nested Monte-Carlo Search

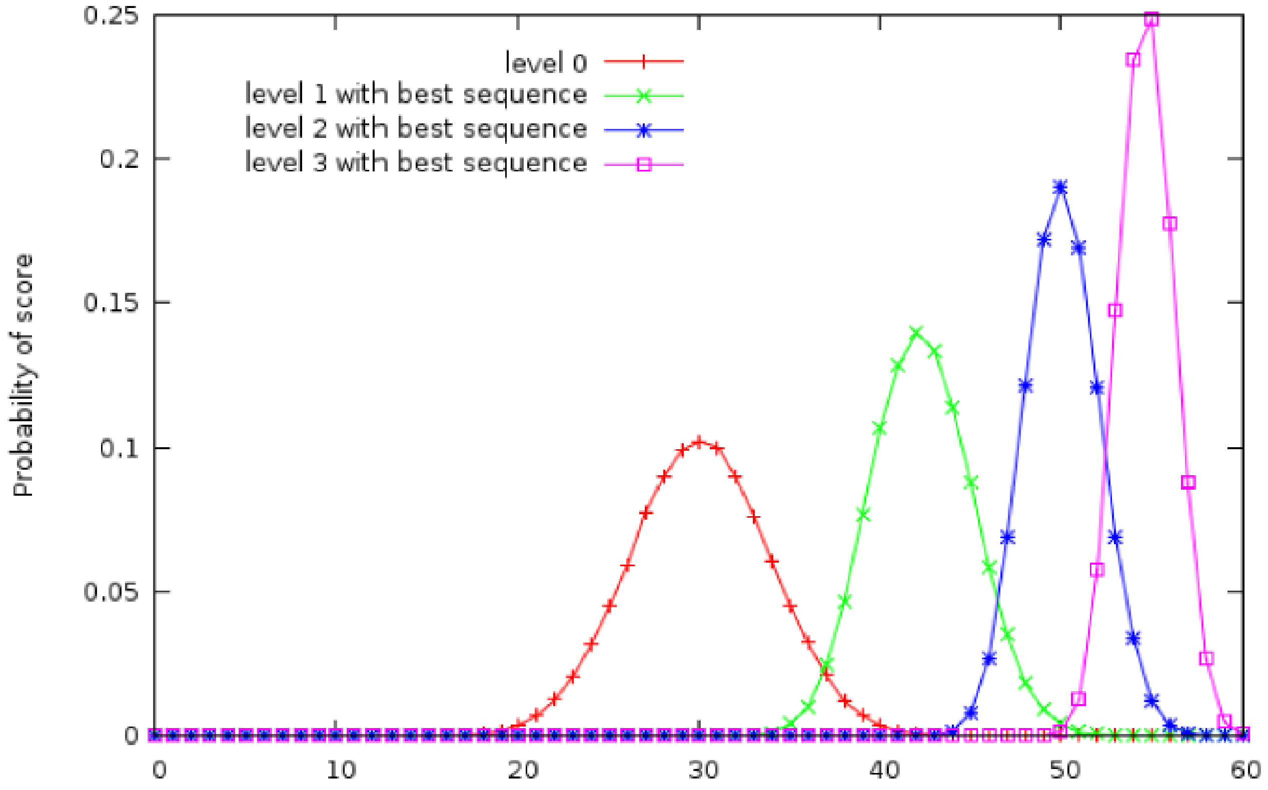
- Play random games at level 0
- For each move at level n+1, play the move then play a game at level n
- Choose to play the move with the greatest associated score
- Important : memorize and follow the best sequence found at each level

NMCS PSEUDO-CODE

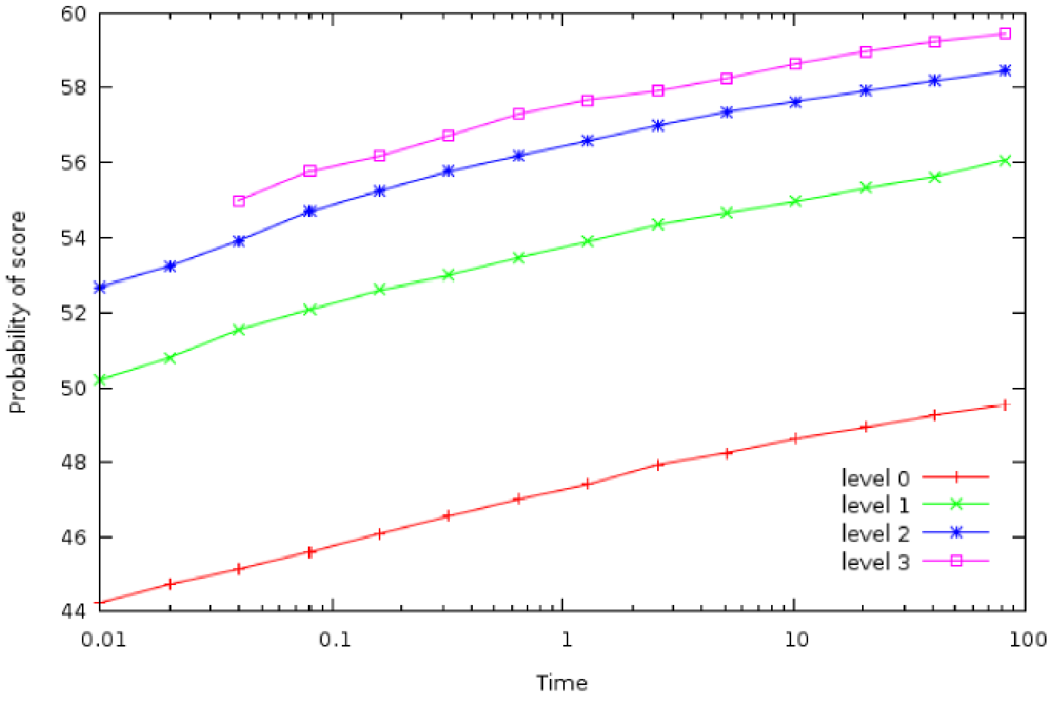
Algorithm 4 The NMCS algorithm.

```
NMCS (state, level)
if level == 0 then
    return playout (state, uniform)
end if
BestSequenceOfLevel ← ∅
while state is not terminal do
    for m in possible moves for state do
        s ← play (state, m)
        NMCS (s, level − 1)
        update BestSequenceOfLevel
    end for
    bestMove ← move of the BestSequenceOfLevel
    state ← play (state, bestMove)
end while
```

TYPICAL NMCS



- Mean score in real time



TSP

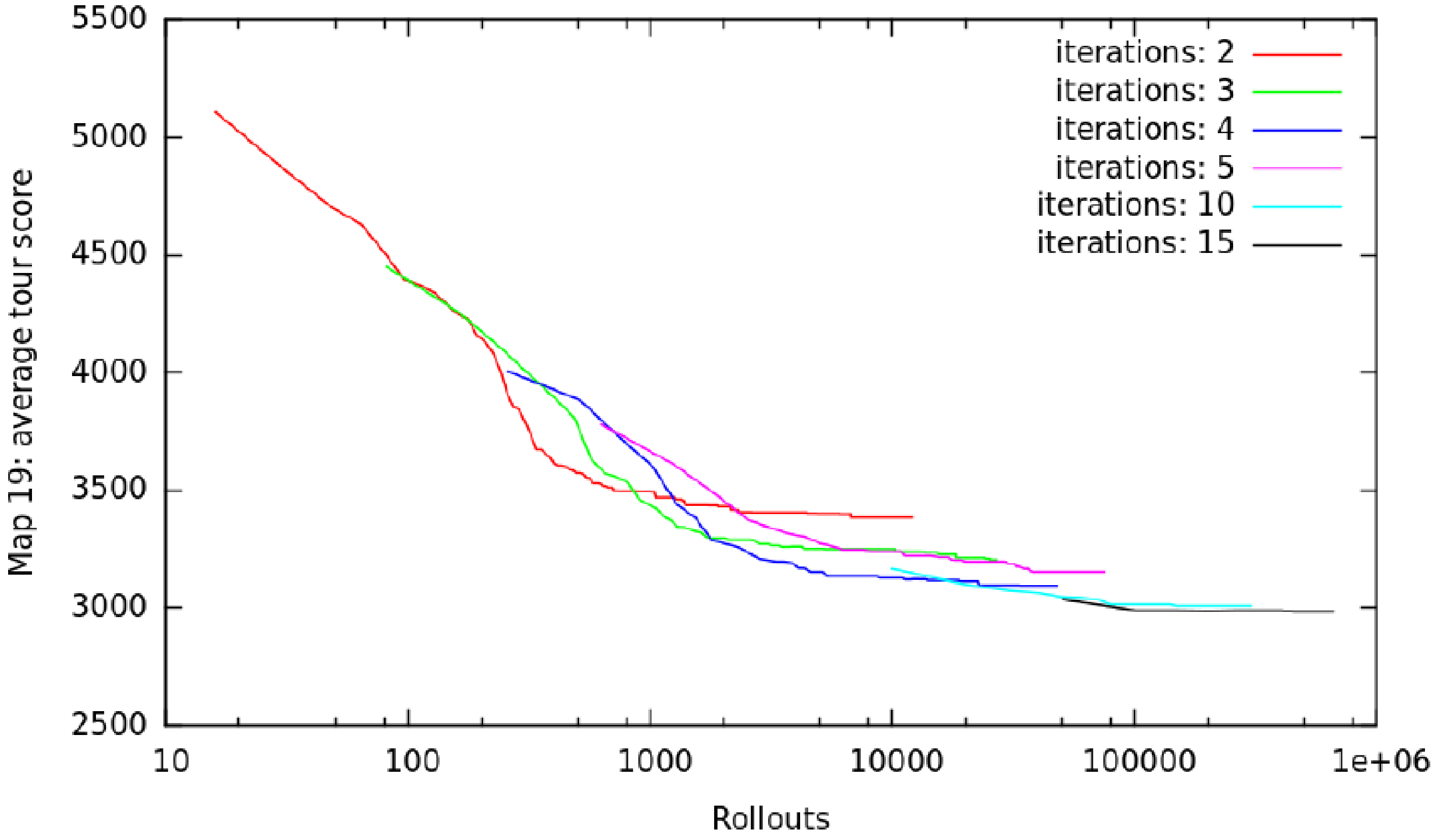
Given a map, compute a *minimum-cost* round trip visiting certain cities

Graph reduction: Precompute *all-pairs-shortest-paths*

Given a distance matrix, compute a *minimum-cost* trip

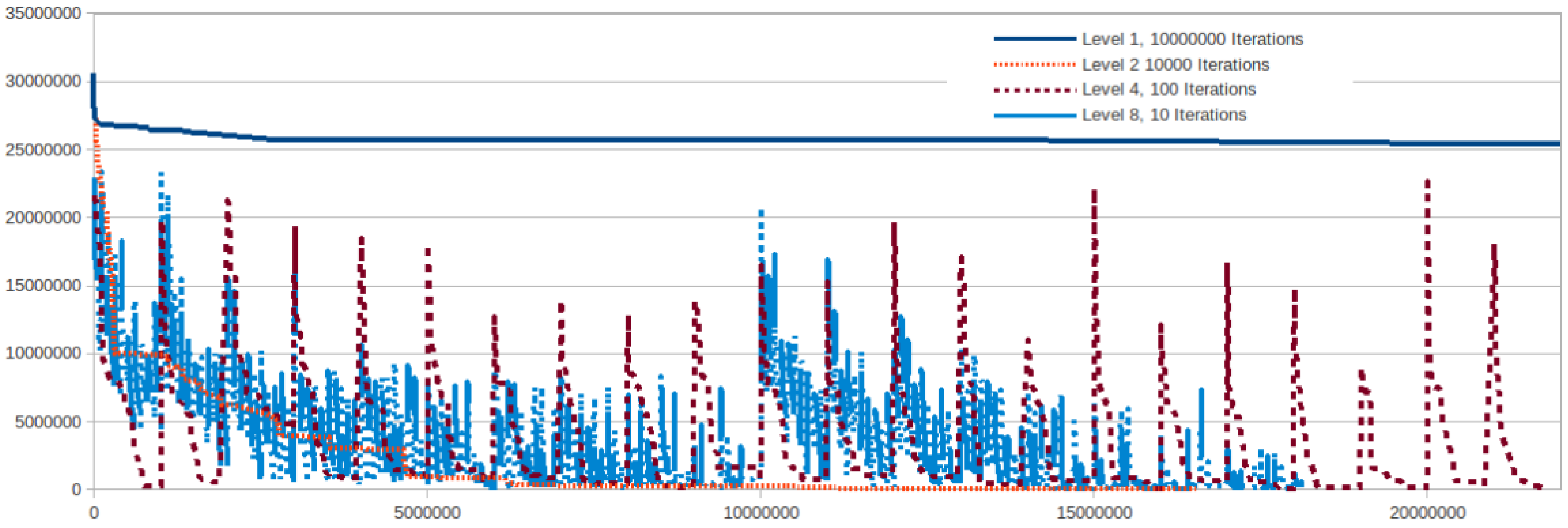
- *Model problem as an IP and call solver* (CPLEX, IPSolve,...)
- *Neighborhood search* (xOPT: SA; GA; AA; PSO; LNS,...)
- *(Depth-First) Branch and Bound* with
  - **DFBnB$_0$** No Heuristic – incremental $O(1)$ time
  - **DFBnB$_1$** Column/Row Minima – incremental $O(1)$ time
  - **DFBnB$_2$** Assignment Problem – incremental $O(n^2)$ time
- ...

LEARNING CURVE



NESTED ROLLOUT POLICY ADAPTATION

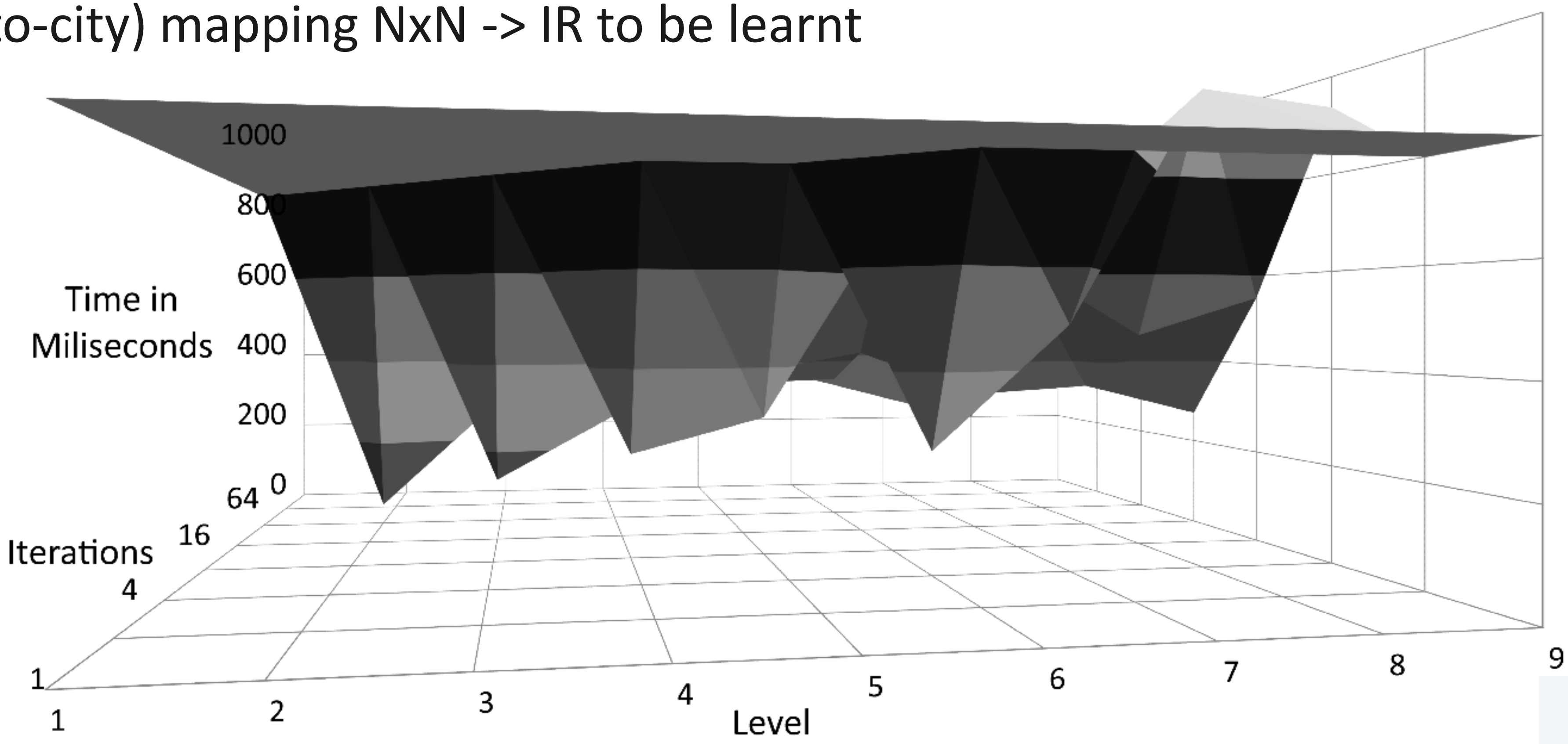
Input: Iteration width (exploitation), nestedness (exploration)
Policy: (city-to-city) mapping NxN -> IR to be learnt



NESTED ROLLOUT POLICY ADAPTATION

Input: Iteration width (exploitation), nestedness (exploration)

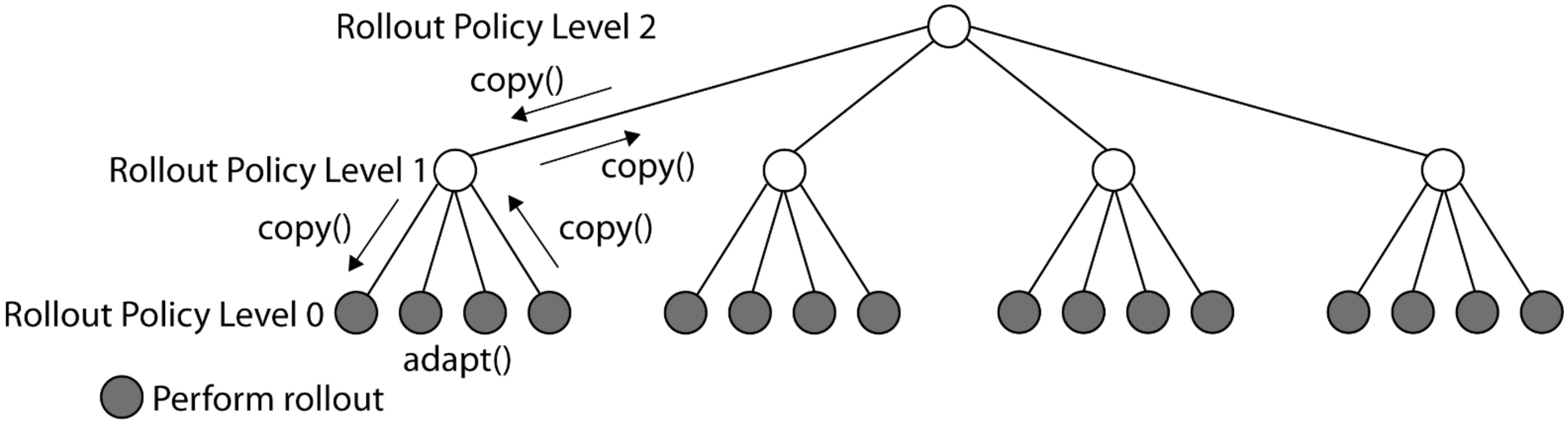
Policy: (city-to-city) mapping NxN -> IR to be learnt



AI CENTER
FEE CTU

NESTED ROLLOUT POLICY ADAPTATION

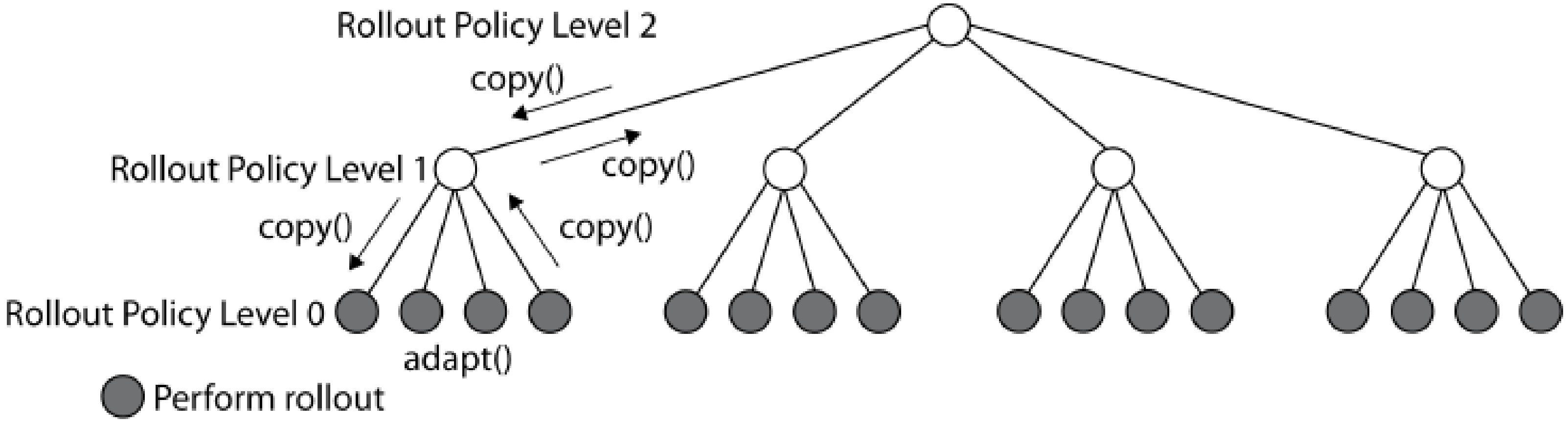
Input: Iteration width (exploitation), nestedness (exploration)
Policy: (city-to-city) mapping NxN -> IR to be learnt



AI CENTER
FEE CTU

NRPA

```
NRPA(level)
    best.score = MAX;
    if (level == 0) best = ROLLOUT();
    else
      backup[level] = global;
      for (i=0; i<iterations; i++)
        current = NRPA(level-1);
        if (current.score < best.score)
          best = current; ADAPT(best);
      global = backup[level];
    return best;
```



AI CENTER
FEE CTU

PLAYOUT

Algorithm 1 The playout algorithm

```
playout (state, policy)
    sequence ← []
    while true do
        if state is terminal then
            return (score (state), sequence)
        end if
        z ← 0.0
        for m in possible moves for state do
            z ← z + exp (policy [code(m)])
        end for
        choose a move with probability exp(policy[code(move)]) / z
        state ← play (state, move)
        sequence ← sequence + move
    end while
```


AI CENTER
FEE CTU

ADAPT

Algorithm 2 The Adapt algorithm

```
Adapt (policy, sequence)
    polp ← policy
    state ← root
    for move in sequence do
        polp [code(move)] ← polp [code(move)] + α
        z ← 0.0
        for m in possible moves for state do
            z ← z + exp (policy [code(m)])
        end for
        for m in possible moves for state do
            polp [code(m)] ← polp [code(m)] - α * exp(policy[code(m)]) / z
        end for
        state ← play (state, move)
    end for
    policy ← polp
```

SEARCH

Algorithm 3 The NRPA algorithm.

```
NRPA (level, policy)
    if level == 0 then
        return  playout (root, policy)
    else
        bestScore ← −∞
        for N iterations do
            (result,new) ← NRPA(level − 1, policy)
            if result ≥ bestScore then
                bestScore ← result
                seq ← new
            end if
            policy ← Adapt (policy, seq)
        end for
        return  (bestScore, seq)
    end if
```

THEORY...

The probability p_{ik} of choosing the move m_{ik} in a playout is the softmax function:

$$p_{ik} = \frac{e^{w_{ik}}}{\Sigma_j e^{w_{ij}}}$$

The cross-entropy loss for learning to play move m_{ib} is $C_i = -log(p_{ib})$. In order to apply the gradient we calculate the partial derivative of the loss: $\frac{\delta C_i}{\delta p_{ib}} = -\frac{1}{p_{ib}}$. We then calculate the partial derivative of the softmax with respect to the weights:

$$\frac{\delta p_{ib}}{\delta w_{ij}} = p_{ib}(\delta_{bj} - p_{ij})$$

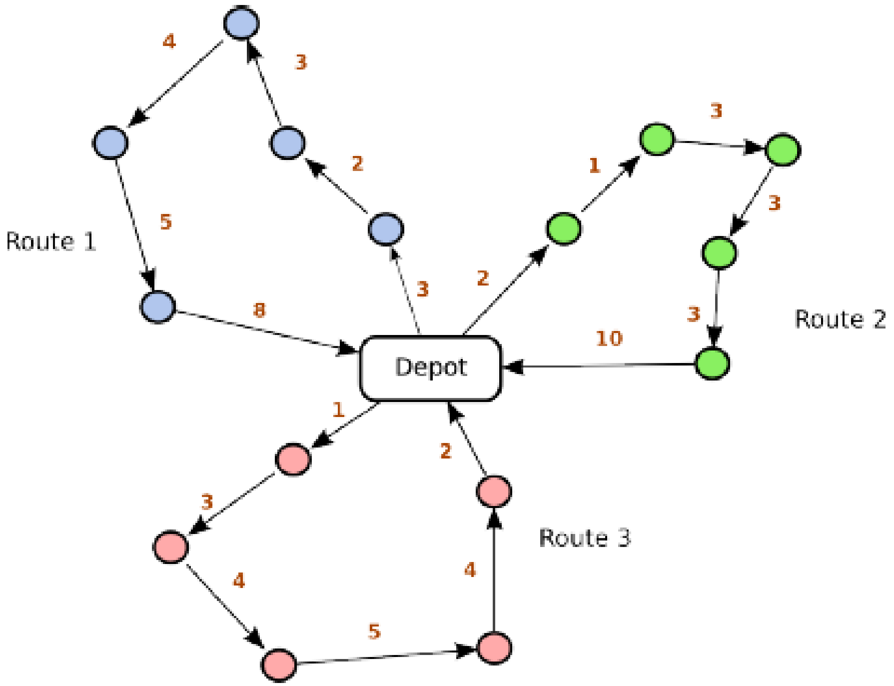
Where $\delta_{bj} = 1$ if $b = j$ and 0 otherwise. Thus the gradient is:

$$\nabla w_{ij} = \frac{\delta C_i}{\delta p_{ib}} \frac{\delta p_{ib}}{\delta w_{ij}} = -\frac{1}{p_{ib}} p_{ib}(\delta_{bj} - p_{ij}) = p_{ij} - \delta_{bj}$$

If we use α as a learning rate we update the weights with:

$$w_{ij} = w_{ij} - \alpha(p_{ij} - \delta_{bj})$$

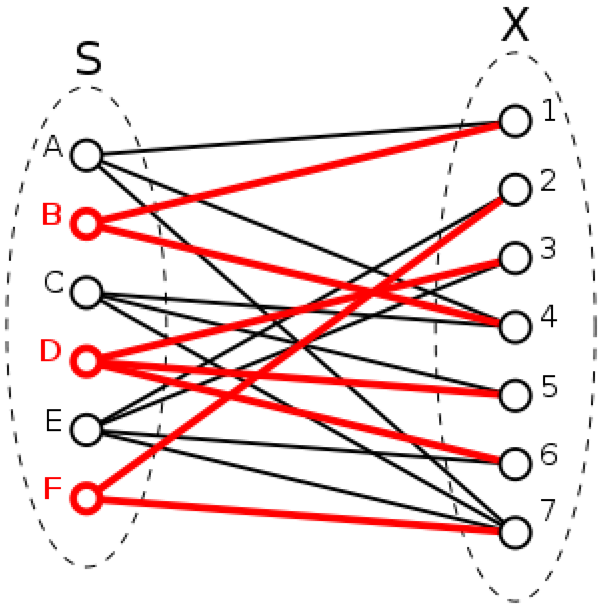
MONTE-CARLO SEARCH FOR VRP



- No additional knowledge needed: rollout function fits in paper.
- Solving Solomon's 100-city problem R101 optimally in 10min (score: 1650.7823, rollouts: 3240000), new highscore in R102
- Very particular benchmark: Even number precision matters.
- Solving Homberger's r2_2_4 (200 packages, 4 vehicles), r2_2_8 (200 packages, 8 vehicles), and r2_4_4 (400 packages, 4 vehicles) in 2h

MONTE-CARLO SEARCH 4 HITTING SET

Given a bipartite graph $G = (V, E)$ with $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, and $E \subseteq (V_1 \times V_2)$, find a subset V' of V_1 of minimal cardinality, so that all nodes in V_2 are covered, i.e., for every $v_2 \in V_2$ there is a $v_1 \in V_1$ such that $(v_1, v_2) \in E$.

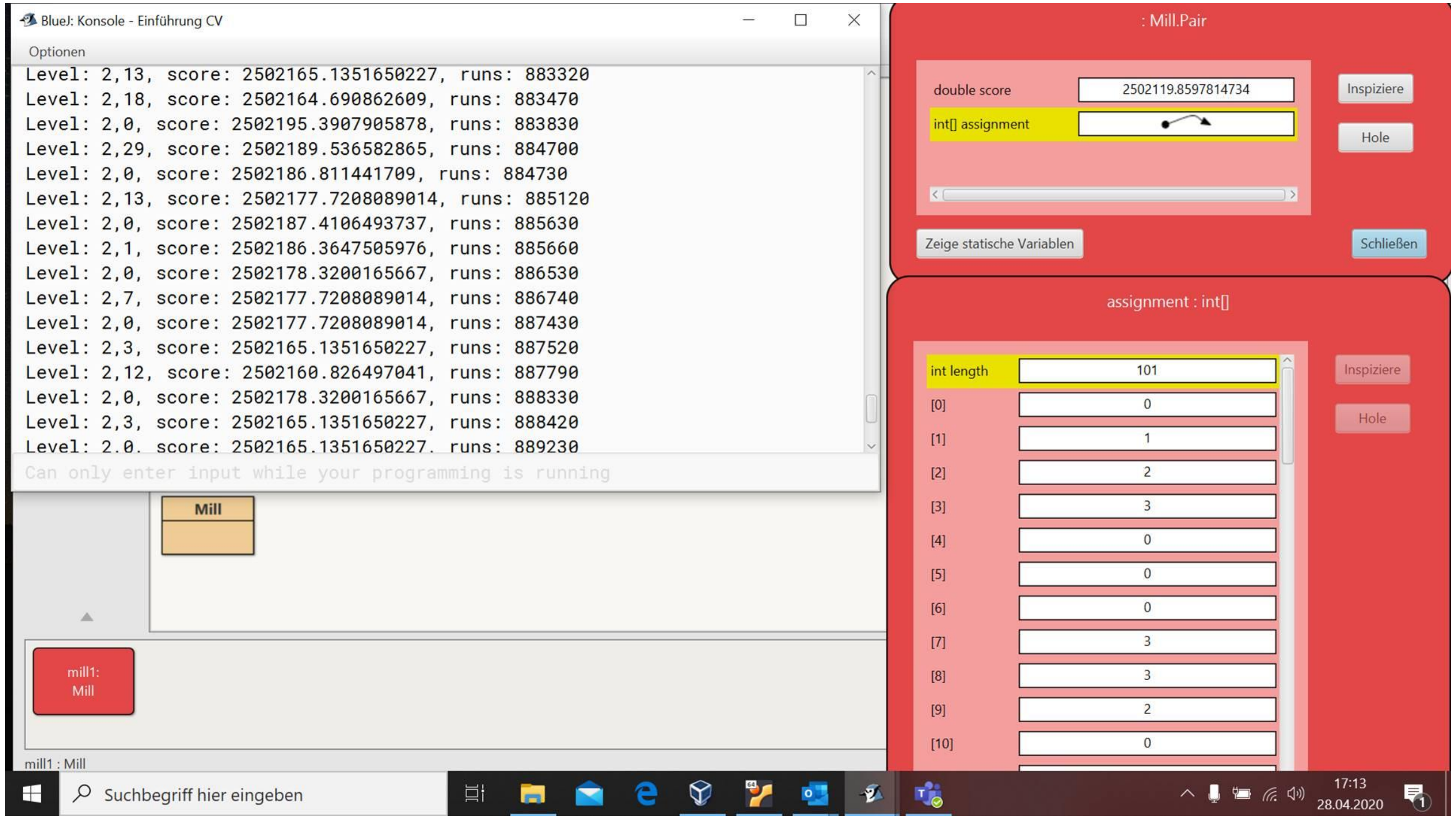



```
class Game {
 public:
  int length, size;
  Move rollout[MaxPlayoutLength];
  Game () {
   for (int j=0; j<SET; j++) visited[j] = 0;
   for (int j=0; j<HITTING; j++) chosen[j] = 0;
   length = size = 0;
  }
  int code (Move m) { return m; }
  bool terminal() { return size >= SET*ALPHA; }
  double score() { return length - size*100; }
  void play(Move m) {
    rollout[length++] = m;
    chosen[m] = 1;
    for (int j = 0; j<SET; j++)
      if (visited[j] == 0 && adjacent[m][j]) {
        visited[j] = 1;
        size++;
      }
  }
  int legalMoves(Move moves[MaxLegalMoves]) {
    int succs = 0;
    for (int m = 0; m < HITTING; m++)
      if (chosen[m] == 0) moves[succs++] = m;
    return successors;
  }
};
```

AI CENTER
FEE CTU

PRAXIS...

https://nms.kcl.ac.uk/stefan.edelkamp/lectures/pi1/programs/VRP.java



SUMMARY

Monte Carlo Search is a simple algorithm that gives state of the art results for multiple problems:

– Games

– Puzzles

– Snake in the box

– Pancake

– Logistics

– Multiple Sequence Alignement

– RNA Inverse Folding