Artificial Intelligence

Local and Stochastic Search

Prof. Dr. habil. Jana Koehler

Dr. Sophia Saller, M. Sc. Annika Engel

Summer 2020

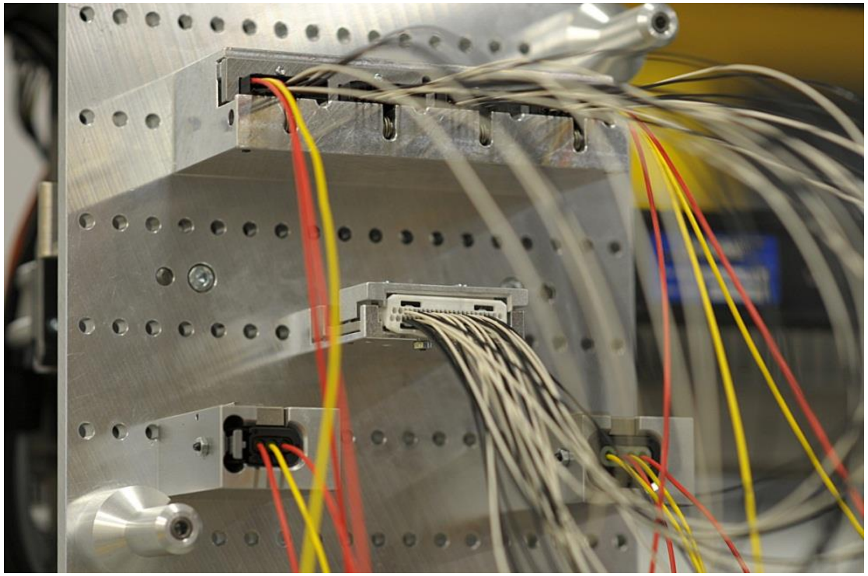
Agenda

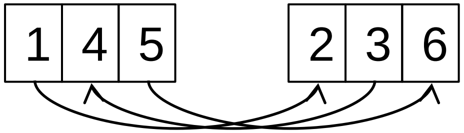
- Searching very large search spaces
- Local extrema & plateaus
- Randomized search strategies
  – Random restarts and moves
  – Tabu search
- Algorithms
  (1) Hill climbing
  (2) Simulated Annealing
  (3) UCT
  (4) Genetic Algorithms
  (5) Ant Colony Optimization

Recommended Reading

- AIMA Chapter 4: Beyond Classical Search
  - 4.1 Local Search Algorithms and Optimization Problems
    - 4.1.1 Hill-climbing search
    - 4.1.2 Simulated annealing
    - 4.1.4 Genetic algorithms
- Papers:
  - A Survey of Monte Carlo Tree Search Methods
    C. Browne et. al.
    IEEE Transactions on Computational Intelligence and AI in games (2012)
  - Finite-time Analysis of the Multiarmed Bandit Problem
    P. Auer, N. Cesa-Bianchi, P. Fischer
    Machine Learning 47.2-3 (2002): 235-256.
  - Bandit based Monte-Carlo Planning
    Levente Kocsis and Csaba Szepesvári
    European conference on machine learning. Springer *(2006)*

Searching Very Large Search Spaces





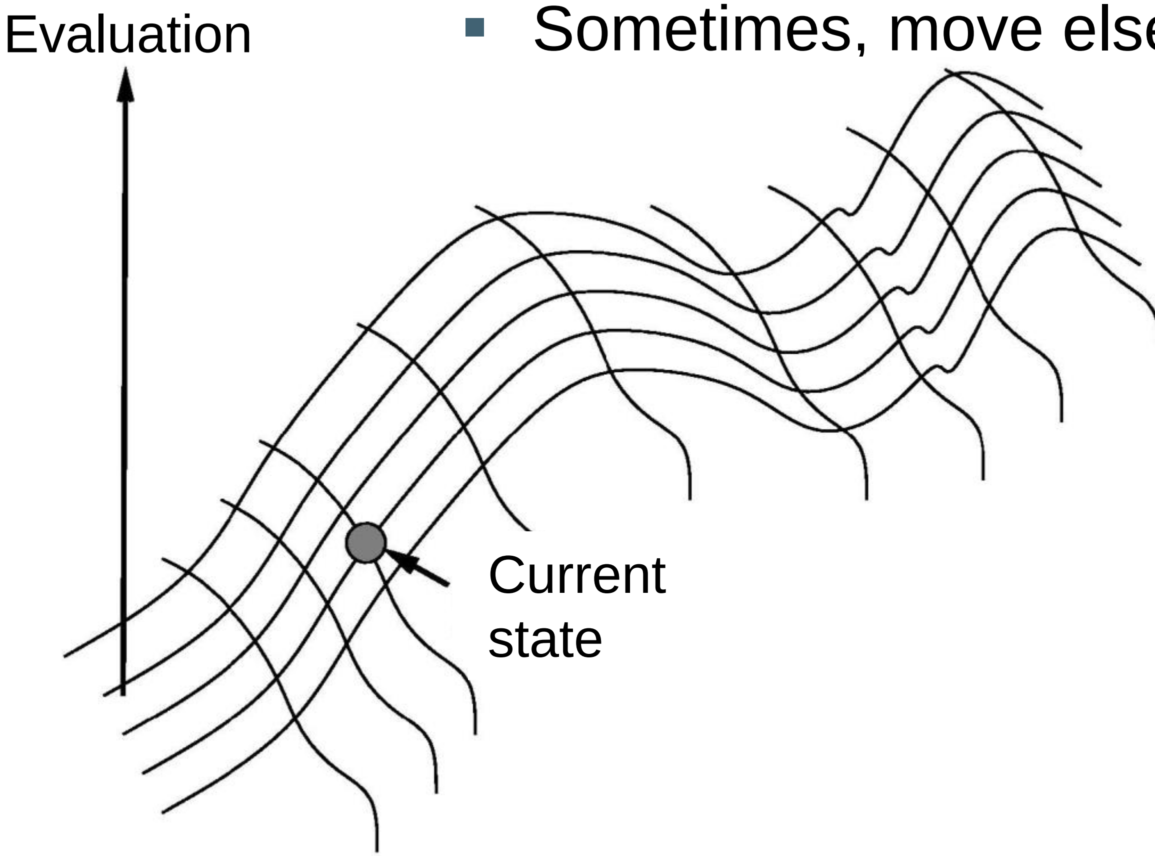
- For n cables, we have $(2n)!$ potential insertion orders

  $\boldsymbol{n=2}$: $4! = 24$, $\boldsymbol{n=40}$: 10^{120}

- No chance to systematically or heuristically explore such spaces!

Basic Idea of Local Search

- Start somewhere in the search space
- Use an evaluation function for each node
- Move towards better evaluated nodes
- Sometimes, move elsewhere



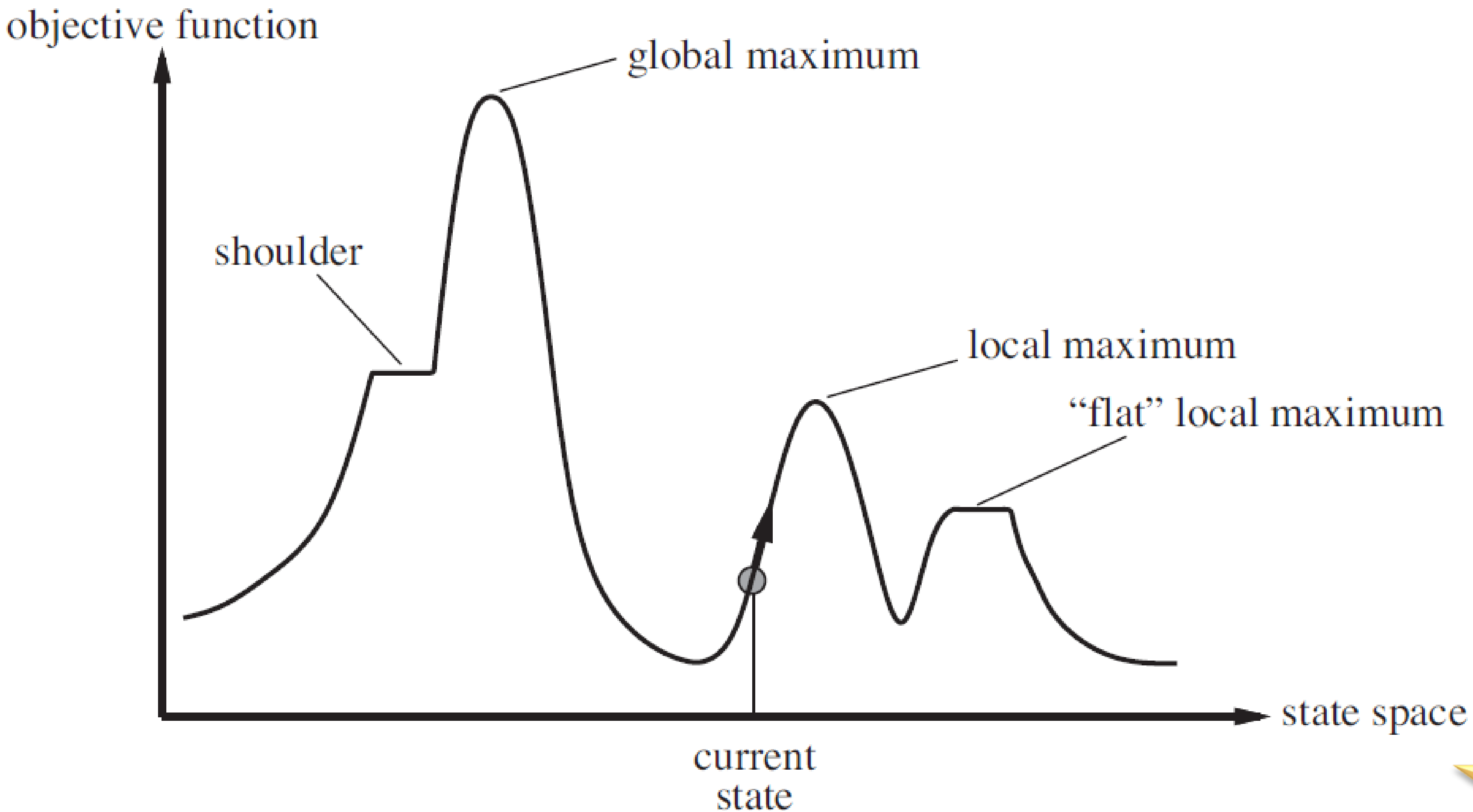
(1) Hillclimbing (Steepest Ascent Search, Greedy Local Search)

function HILL-CLIMBING(problem) **returns** a state that is a local maximum
  current ← MakeNode(*problem*.InitialState)
  loop do
    neighbor ← a highest-valued successor of *current*
    if *neighbor*.Value ≤ *current*.Value **then return** *current*.State
    current ← *neighbor*

- Neither complete nor optimal
- Time complexity: Stops once no better evaluated neighbor can be found (or it encounters a time out)
- Space complexity: $O(b)$ (current state + neighbors)
- In practice, can find good solutions very fast

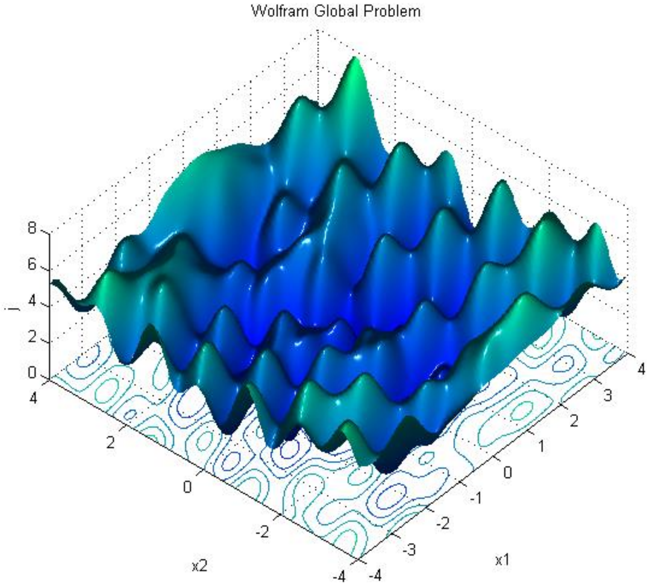
State-Space Landscape

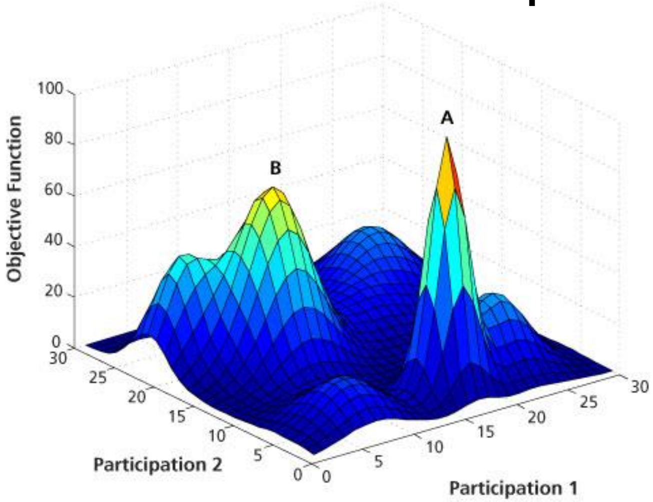
- Plateaus, ridges, local maxima

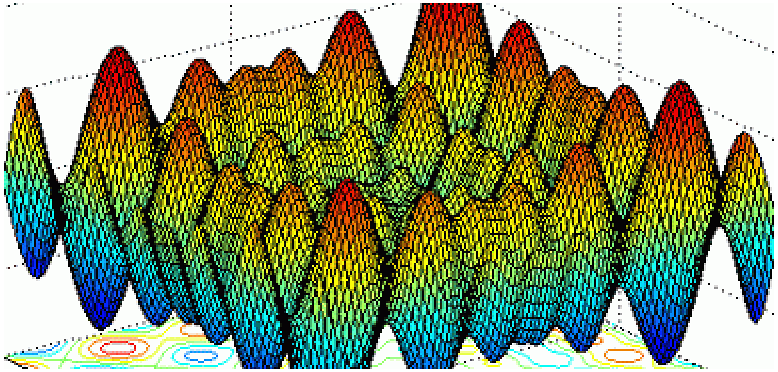


Local Maxima and Minima

- Trap the algorithms in nodes with suboptimal solutions
  – once in such a node, all successors have poorer evaluations

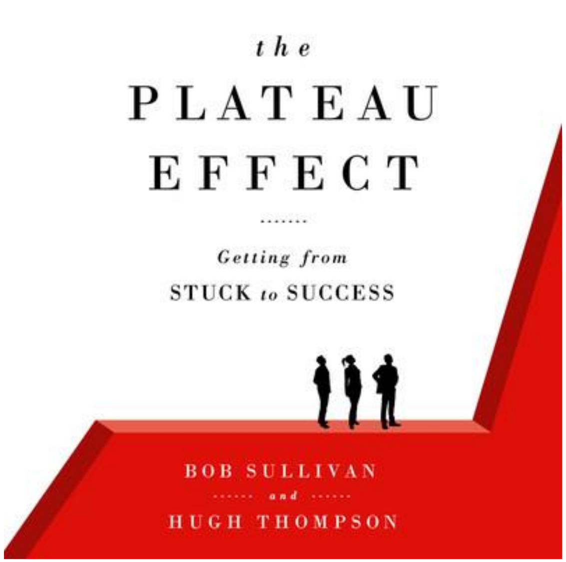






Plateaus

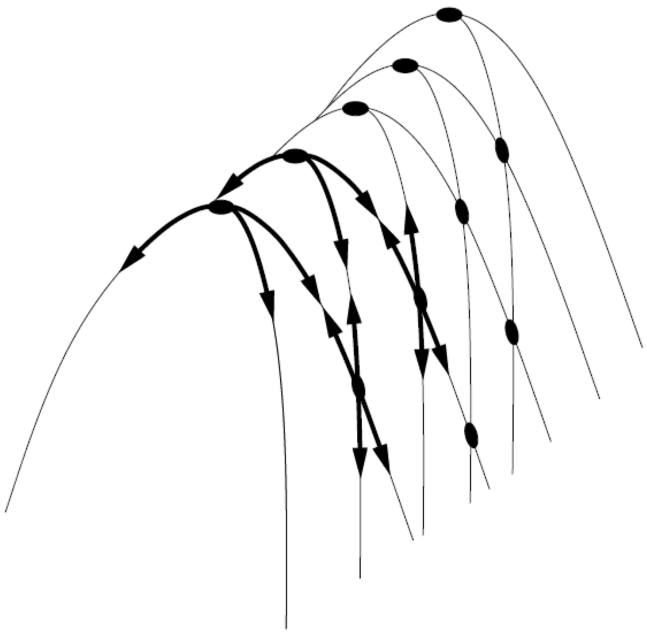
- Cause the algorithm to wander around without any direction
  – all nodes have equally good evaluations



how to escape a plateau when learning, training sports, ….

Ridges

- A sequence of local maxima not directly connected to each other



Escape Techniques

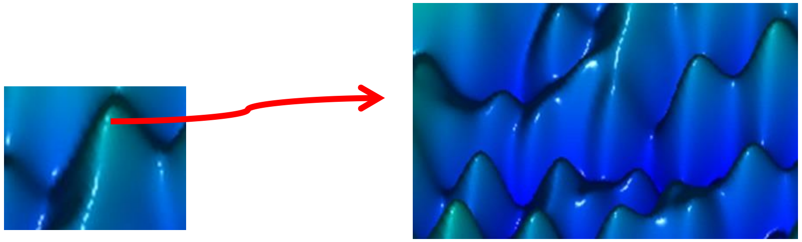
- Tabu Search: Add a memory to a local search algorithm to remember certain moves
  - keep a list of forbidden (visited) states
  - avoid moves that lead to previously explored regions of the search space
    - short term memory: do not reverse a previous move
  - update this list while search progresses
- Random Restart: Start over when no progress is made
  - do a random restart from a randomly generated initial state performing many hill climbing searches
    - theoretically complete, because it will eventually generate the goal state as an initial state
- Random Walk: “Inject noise” = pick a worse or equal evaluated node with a certain probability

Sucess of Escape Strategies

- Which strategies and parameters are successful depends on the
  - problem class and
  - the structure of the search space

- Few local maxima and plateaus, random restart hillclimbing finds good solutions very quickly

- Most difficult (NP-hard) problems have an exponential number of local maxima



Searching the Solution Space

- In many applications, we do not care about the path to the solution
  - 8 queens: the correct placement of queens on the board
  - cable tree wiring: a robust and fast insertion order
  - vacuum world: a plan that cleans all rooms

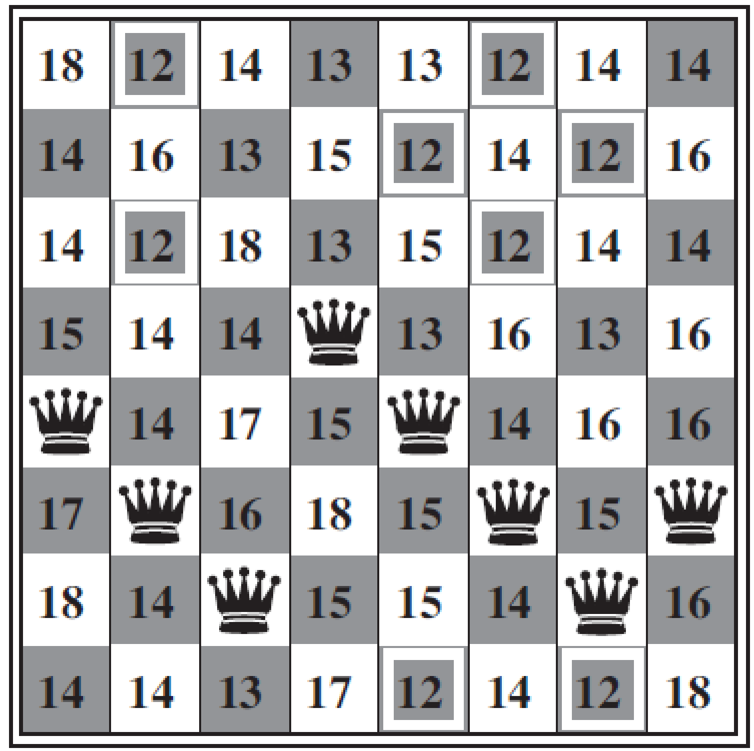
- We can start with some randomly generated (partial) solution and try to improve it
  - take a solution node
  - generate its neighbors (if they have better evaluations)
  - do not keep information about the search path

Hillclimbing 8 Queens

- State: distribution of all 8 queens, one in each column
- h: number of pairs of queens attacking each other
- Successor: select a column and move the queen to another square in the same column

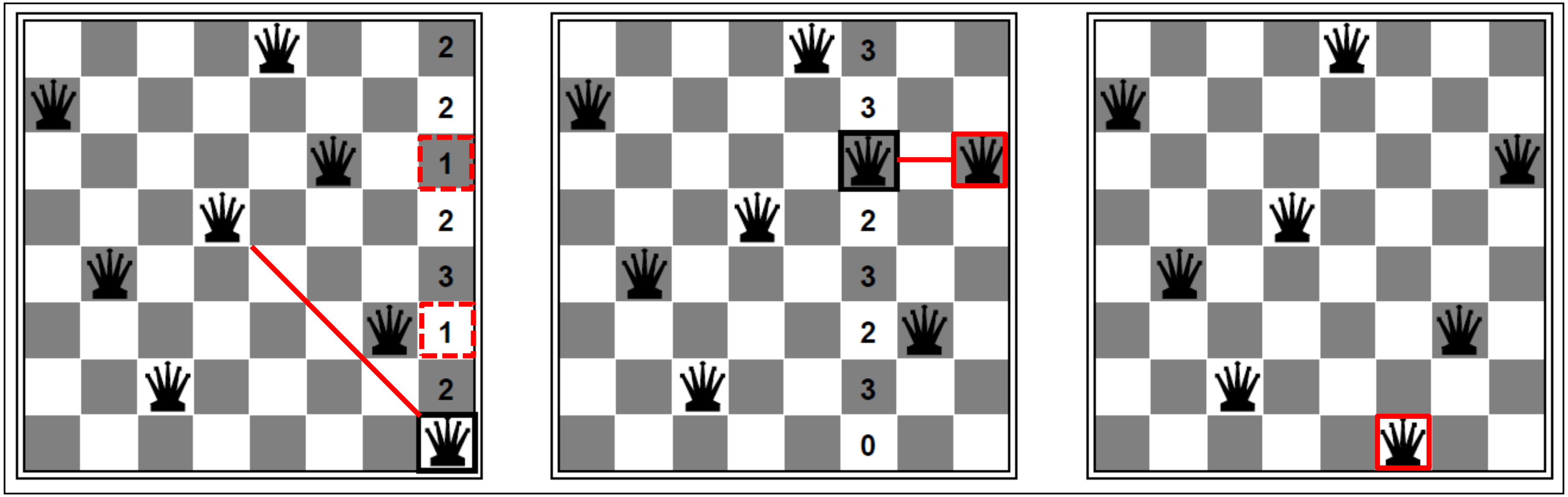
$h = 17$
the best successors have
$h = 12$

hill climbing chooses randomly among the best successors



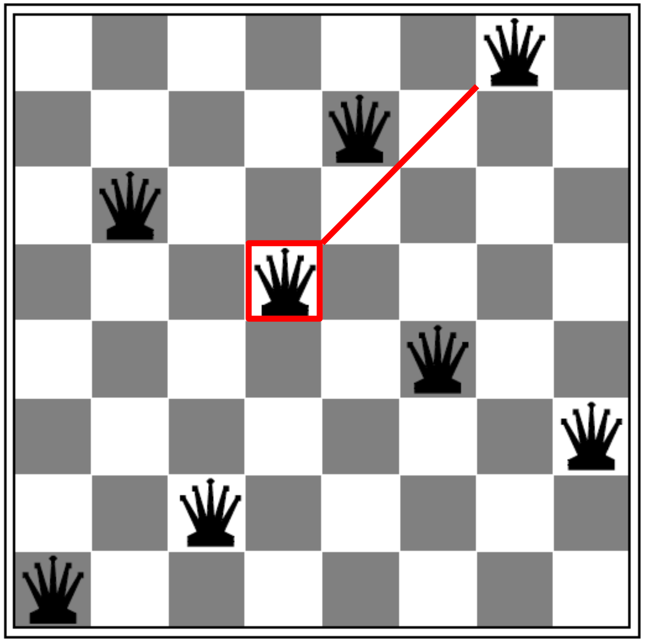
Another Hill Climbing Strategy

- Select a column and move the queen to the square with the fewest conflicts



Local Minimum in 8 Queens

- 19 possible moves
- $h = 1$, no possible move can decrease h



- problem has 10^{14} states (random distributions of the 8 queens on an 8x8 board)
- on randomly generated instances, hillclimbing needs on average 4 steps to find a solution, 3 to get stuck in local minimum

➢ Hillclimbing gets stuck on 86% of all 8 queens problems and can solve 14%

Random Restarts and Random Walks on 8 Queens

- Success rate of a single run of hillclimbing on 8 queens
  - p=14%, take 1/0.14 = 7.14 restarts
  - finds a solution under a minute for 3 million queens

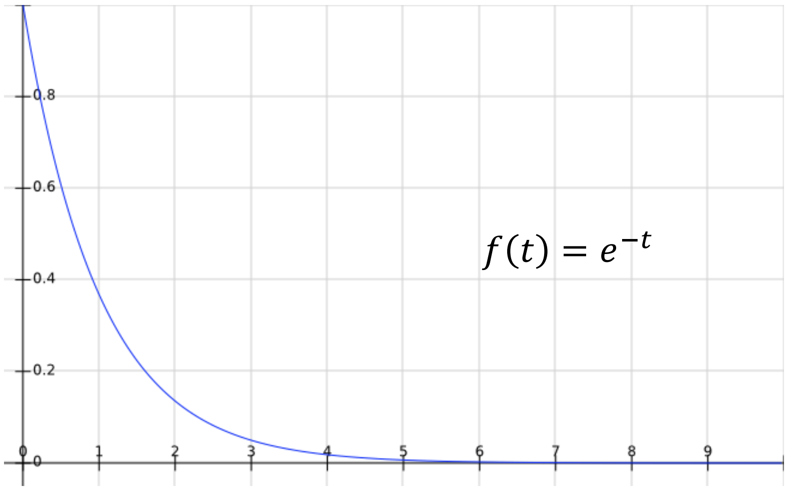
- Add up to 100 sideway moves (to nodes with equal evaluation) to 8 queens in a singe run of the algorithm
  - hillclimbing can then solve 94% of all instances
  - In average, requires 21 steps for a solution and 64 steps for a failure

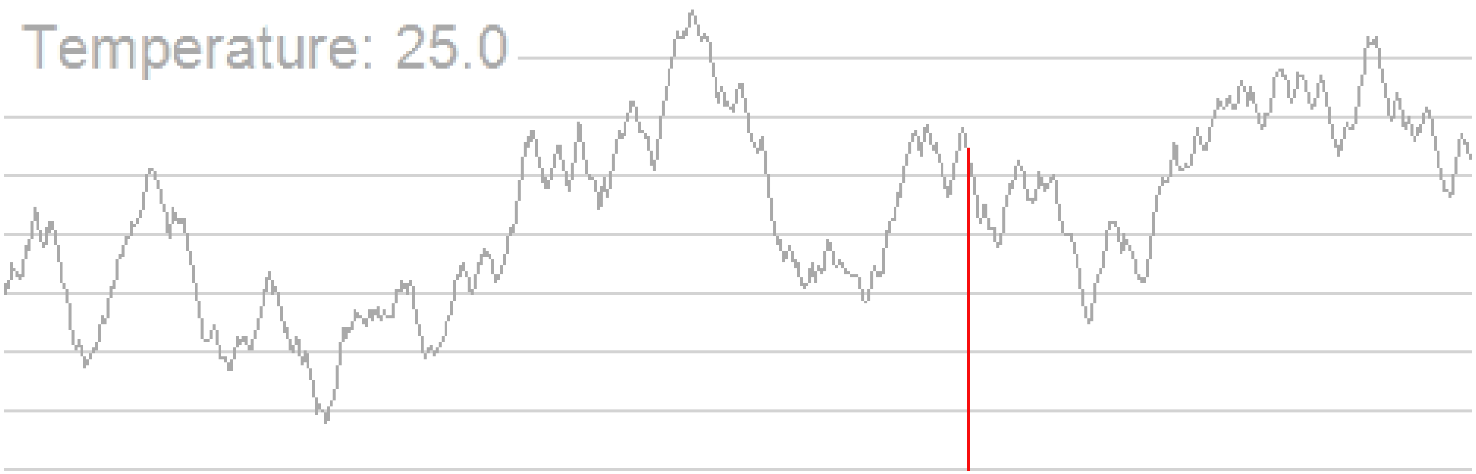
➢ Successful local search algorithms combine randomness (exploration) with following the heuristic (exploitation)

(2) Simulated Annealing

- T (a „temperature“) gradually decreases (cools down)
- Slow decrease in probability of accepting worse solutions

function SIMULATED-ANNEALING(*problem*) **returns** a solution state
  $current \leftarrow$ MakeNode(*problem*.InitialState)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow \frac{1}{t}$
    if $T = 0$ **then return** *current*
    $next \leftarrow$ a randomly selected successor of *current*
    $\Delta V \leftarrow$ *current*.Value - *next*.Value
    if $\Delta V < 0$ **then** $current \leftarrow next$
    else with probability $e^{-\Delta V \cdot t}$ do $current \leftarrow next$





https://commons.wikimedia.org/w/index.php?curid=25010763

(3) UCT: A Stochastic Search Algorithm

- L. Kocsis and C. Szepesvári: Bandit based Monte-Carlo Planning, *European Conference on Machine Learning*, 2006

Monte Carlo Tree Search (MCTS) + Upper Confidence Bounds (UCB)

⇩

Upper Confidence Bounds for Trees (UCT)

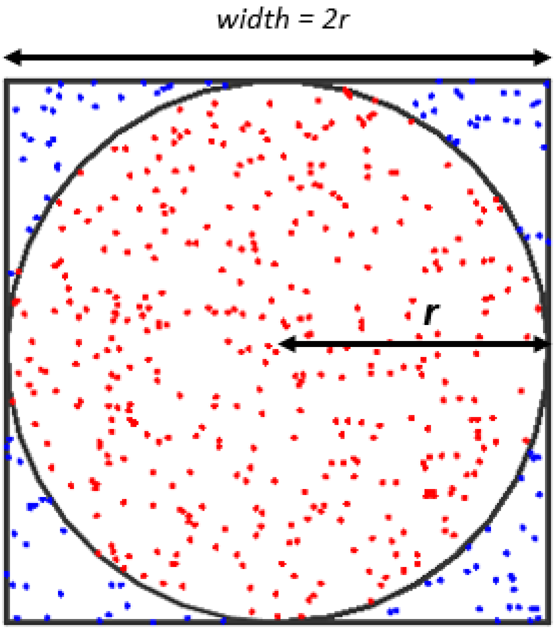
Monte Carlo Algorithms

- Perform repeated random sampling to determine numerical estimations of unknown parameters
  - developed by Stanislaw Ulam and John von Neumann in the Manhattan project to run computer simulations for risk analysis in the 1940s

"As the number of identically distributed, randomly generated variables increases, their sample <u>mean</u> (average) approaches their theoretical mean."

Estimating PI by Observing Rain Drops on a Board

- Area of circle is πr^2
- Area of square is $width^2 = (2r)^2 = 4r^2$
- If we divide the area of the circle by the area of the square we get $\pi/4$
- The same ratio can be used between the number of points within the square and the number of points within the circle
- “Law of large numbers”



$$\pi \approx 4 \cdot \frac{number\ of\ points\ in\ the\ circle}{total\ number\ of\ points}$$

https://www.101computing.net/estimating-pi-using-the-monte-carlo-method/

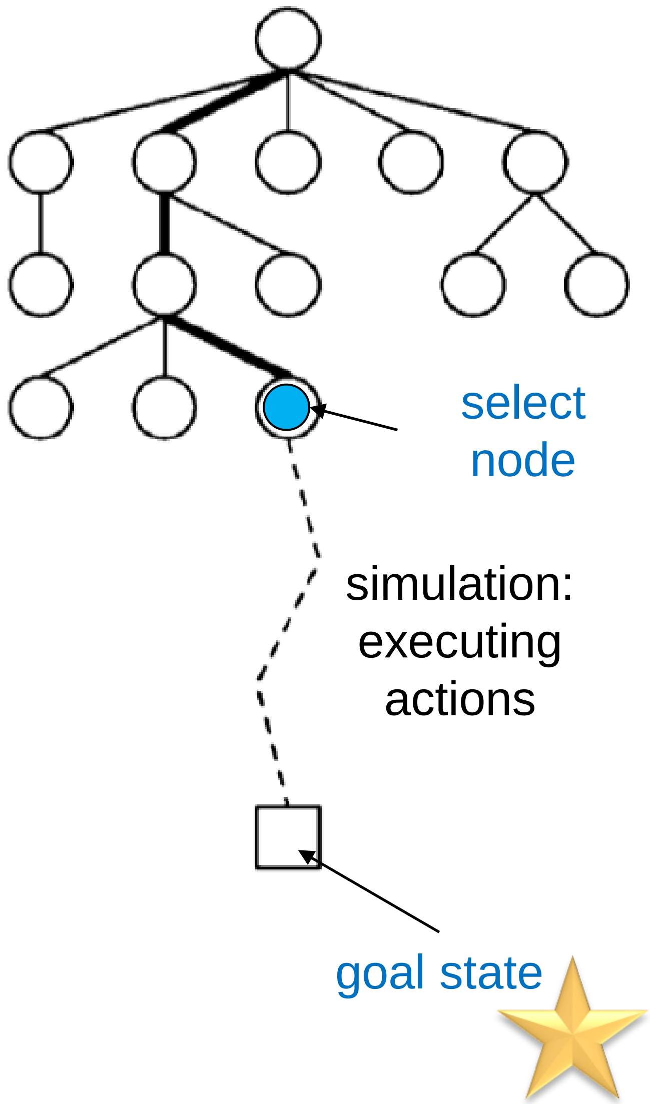
Monte Carlo Tree Search (MCTS)

- Method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results
  - statistical anytime algorithm for which more computing power generally leads to better performance
  - can be used with little or no domain knowledge

- Since the 1990s, Monte Carlo ideas are applied to game playing and planning problems in AI
  - the method of choice for very large search spaces
  - 10^{120} and beyond
  - many variations and improvements exist

The Basic MCTS Process

- A tree is built in an incremental and asymmetric manner:
- For each iteration of the algorithm, a *tree policy* is used to find the next node to be expanded of the current tree
- The tree policy attempts to balance considerations of ***exploration*** (look in areas that have not been well sampled yet) and ***exploitation*** (look in areas which appear to be promising)
- A simulation is run from the selected node and the search tree is updated according to the result in the goal state

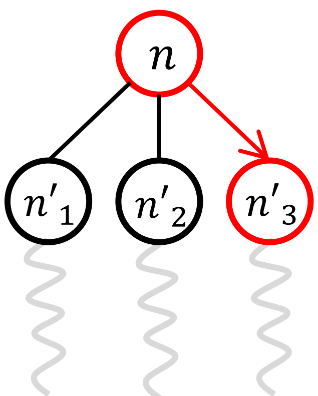
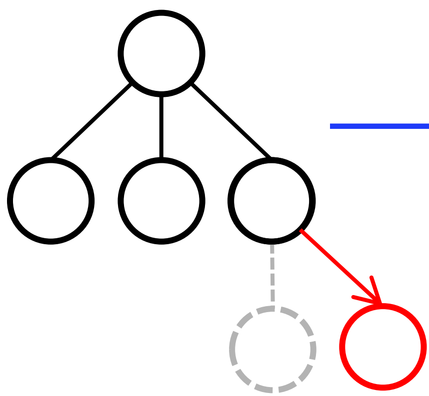
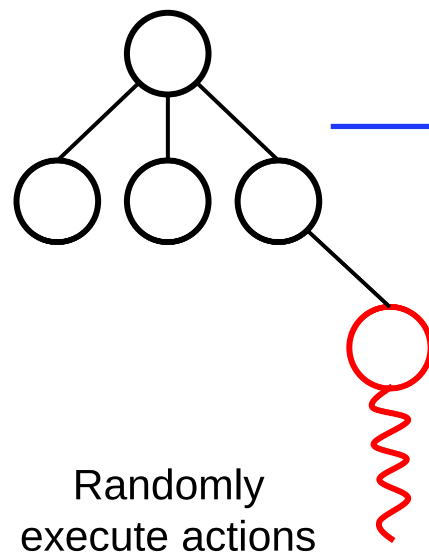
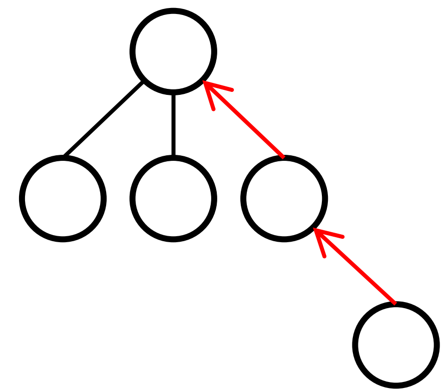


Which Node to Select? ▶ Bandit Problems

- Class of sequential decision problems, in which one needs to choose amongst K actions in order to maximize the cumulative reward by consistently taking the optimal action
  - K arms of a multi-armed bandit slot machine
  - choice of action is difficult as the underlying reward distributions are unknown, and potential rewards must be estimated based on past observations

- Exploitation/Exploration Dilemma
  - need to balance the exploitation of the action currently believed to be optimal with the exploration of other actions that currently appear suboptimal, but may turn out to be superior in the long run
    - Which arm of the bandit to play next?
    - UCB 1 Algorithm
      - Auer et al: Finite-time Analysis of the Multiarmed Bandit Problem, 2002

Overview of Phases in MCTS-based Algorithms

(1) Selection	(2) Expansion	(3) Simulation	(4) Backpropagation
			
Which branch is the most promising once we have tried all at least once?	Expand towards an arbitrary unexplored child in the selected branch	Randomly execute actions until a goal state (or terminal state) is reached	Update the reward towards the root node

Default Policy

Tree Policy

https://www.youtube.com/watch?v=lhFXKNyA0QA
https://www.youtube.com/watch?v=Fbs4lnGLS8M
https://www.youtube.com/watch?v=EGN1KAjtNS4

UCT Algorithm: Overview

function UCTSEARCH(s_0)
  create root node n_0 with
  n_0.State $= s_0$
  $N(n_0) = 0$
  $Q(n_0) = 0$
  while within computational budget **do**
    $n_l \leftarrow$ TREEPOLICY(n_0)
    $r \leftarrow$ DEFAULTPOLICY(n_l.State)
    BACKPROPAGATION(n_l, r)
  return action leading to $\underset{n' \in children\ of\ n_0}{\operatorname{argmax}} N(n')$

Node count

Sum of all rewards of paths through n_0

last node reached during the TreePolicy stage

reward for the goal state reached by running the default policy from state $s(n_l)$

the action a that leads to the best child of the root node n_0
- exact definition of “best” is defined by the implementation

Using UCB1 as Tree Policy

- How to select the next node n' for selection?
  - Take the best UCB1 value

Remember
Q: reward sum
N: visit count

$$\textbf{function } \textsc{Select}(n)$$

$$\textbf{return} \quad \underset{n' \in children\ of\ n}{\textbf{argmax}} \; \frac{Q(n')}{N(n')} + c\sqrt{\frac{2\ln(N(n))}{N(n')}}$$

encourages the ***exploitation*** of higher-reward choices

encourages the ***exploration*** of less visited choices

c is a constant, adjust to lower or increase the amount of exploration

(1)+(2) Selection + Expansion: The Tree Policy

function $\textsc{TreePolicy}(n)$
  while $\text{GoalTest}(n.\text{State}) = \text{FALSE}$ **do**
    if n has unexplored children **then**
      return $\textsc{Expand}(n)$
    else
      $n \leftarrow \textsc{Select}(n)$
  return n

function $\textsc{Expand}(n)$
  choose a in untried actions from $\text{Actions}(n.\text{State})$
  add a new child n' to n with
  $n'.\text{State} = \text{ChildState}(n.\text{State}, a)$
  $N(n') = 0$
  $Q(n') = 0$
  return n'

A node is expandable if it represents a nonterminal state and has unexplored children

(3) Simulation: The Default Policy

- Execution of actions from the selected node until a goal state is reached using a default policy
  - simply applying actions randomly or
  - applying a statistically biased sequence of actions

$$
\begin{array}{l}
\textbf{function } \textsc{DefaultPolicy}(s) \\
\quad \textbf{while } \text{GoalTest}(s) = \text{FALSE } \textbf{do} \\
\qquad \text{choose } a \in \text{Actions}(s) \text{ uniformly at random} \\
\qquad s \leftarrow \text{ChildState}(s, a) \\
\quad \textbf{return } \text{reward for state } s
\end{array}
$$

- Once a goal state is reached, the simulation finishes, the goal state is evaluated and the evaluation is backed up to the ancestors of the selected node
  - No need to evaluate intermediate states!

(4) Backpropagation

- Each node's visit count is incremented, and its Q-value updated
- The reward value may be
  - a discrete (win/draw/loss) result or
  - a continuous reward value
  - Usually normalized to the interval [0,1]

$$
\begin{array}{l}
\textbf{function } \textsc{Backpropagate}(n, r) \\
\quad \textbf{while } n \text{ is not null } \textbf{do} \\
\qquad N(n) \leftarrow N(n) + 1 \\
\qquad Q(n) \leftarrow Q(n) + r \\
\qquad n \leftarrow \text{Parent of } n
\end{array}
$$

Summary of Algorithm

- Download the complete description of the algorithm in pseudo code from CMS > Materials > Supplementary Materials

AI Vorlesung - UCT Search Algorithm

Dr. Sophia Saller

Algorithm 1 UCT Algorithm

```
function UCTSearch(s_0)                                  ▷ This is the core of the algorithm
    create root node n_0 with                            ▷ Initialise the root node
    n_0.State = s_0
    N(n_0) = 0
    Q(n_0) = 0
    while within computational budget do
        n_l ← TreePolicy(n_0)                            ▷ n_l is last node reached during the Tree Policy stage
        r ← DefaultPolicy(n_l.State)                     ▷ r is reward for terminal state reached in simulation
        Backpropagation(n_l, r)
    return action leading to argmax_{n' ∈ children of n_0} N(n')   ▷ Action leading to “best” child

function TreePolicy(n)                                   ▷ Used to select node to run simulation from
    while GoalTest(n.State) = FALSE do                   ▷ As long as we have not reached a goal state
        if n has unexplored children then
            return Expand(n)
        else
            n ← Select(n)
    return n                                             ▷ Returns node to run next simulation from

function Expand(n)                                       ▷ Expansion
    choose a in untried actions from Actions(n.State)
    add a new child n' to n with                         ▷ Initialise new node
    n'.State = ChildState(n.State, a)
    N(n') = 0
    Q(n') = 0
    return n'

function Select(n)                                       ▷ Selection
    return argmax_{n' ∈ children of n} Q(n')/N(n') + c·sqrt(2 ln(N(n)) / N(n'))   ▷ Returns the child node with best UCB1 value

function DefaultPolicy(s)                                ▷ Simulation
    while GoalTest(s) = FALSE do
        choose a ∈ Actions(s) uniformly at random        ▷ Take random actions until we reach goal state
        s ← ChildState(s, a)                             ▷ Apply a to current state to get to next state
    return reward for state s

function Backpropagate(n, r)                             ▷ Backpropagation
    while n is not null do
        N(n) ← N(n) + 1
        Q(n) ← Q(n) + r
        n ← Parent of n                                  ▷ If n is the root node, its parent is null
```

(1) SELECTION

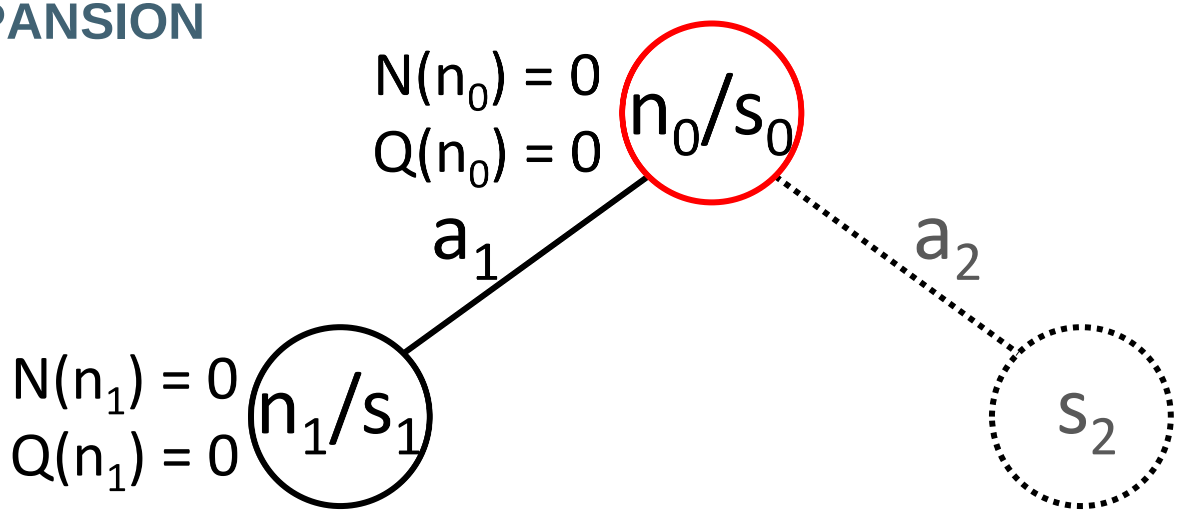
$N(n_0) = 0$

$Q(n_0) = 0$

n_0/s_0

Create the root node and select it

(2) Expansion



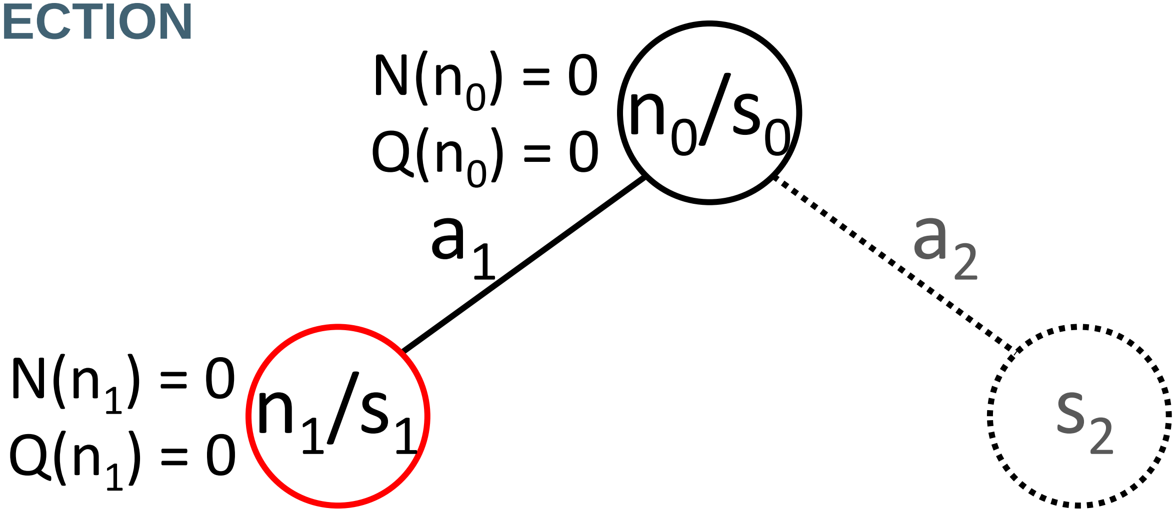
further child states are represented by dotted lines

```
function TreePolicy(n)
    while GoalTest(n.State) = FALSE do
        if n has unexplored children then
            return Expand(n)
        else
            n ← Select(n)
    return n
```

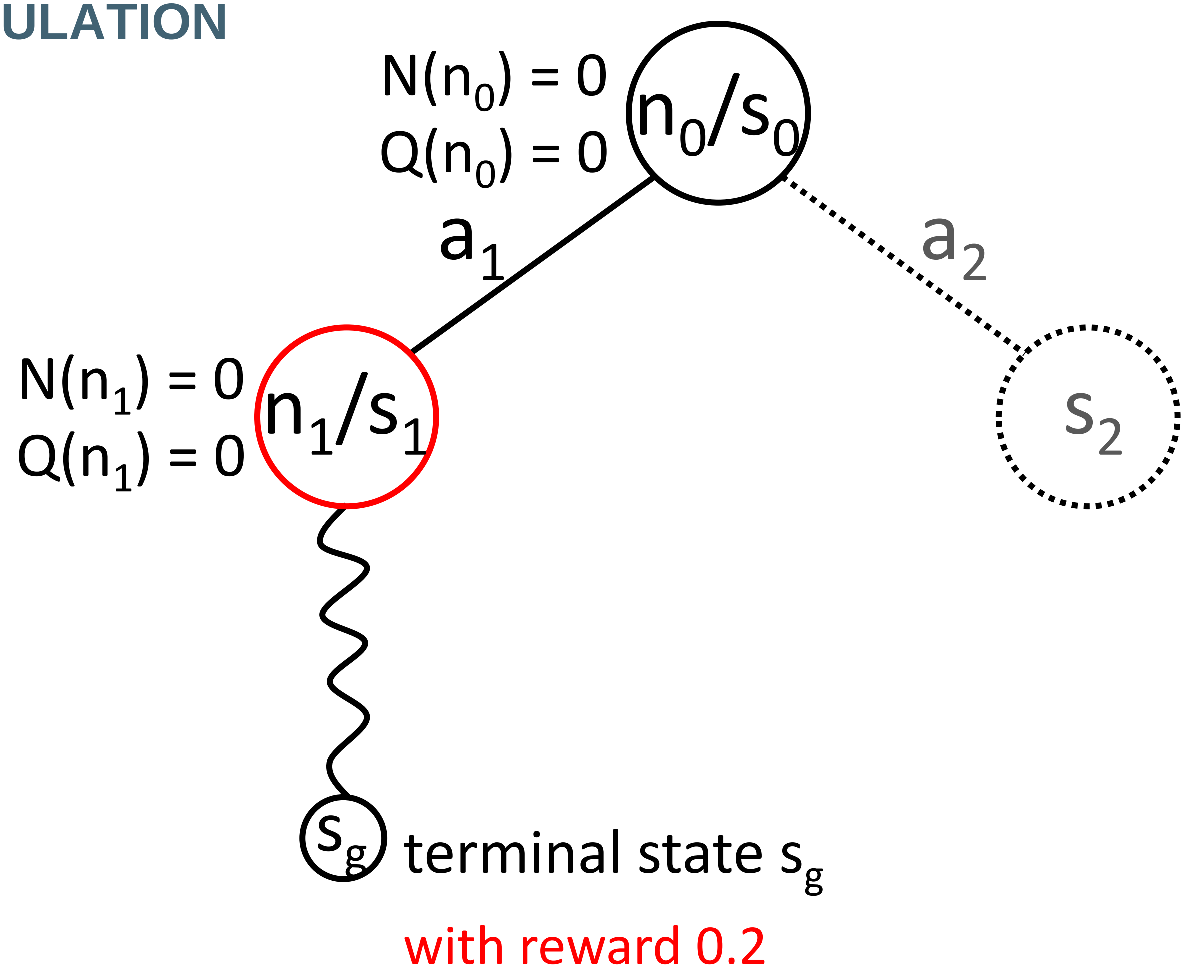
```
function Expand(n)
    choose a in untried actions from Actions(n.State)
    add a new child n′ to n with
    n′.State = ChildState(n.State, a)
    N(n′) = 0
    Q(n′) = 0
    return n′
```

Apply tree policy: n_0 has unexplored children, pick an untried action

(1) SELECTION

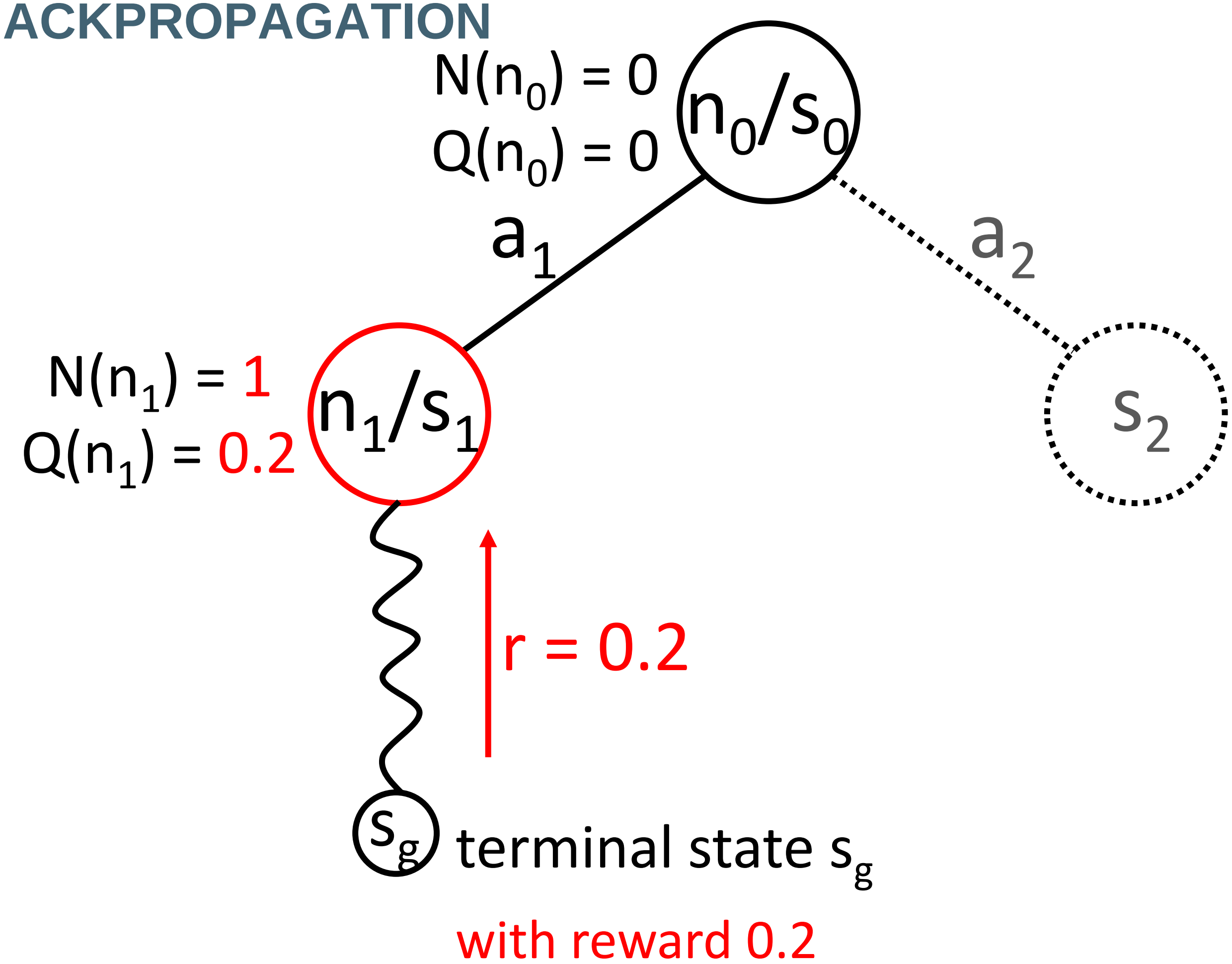


(3) SIMULATION



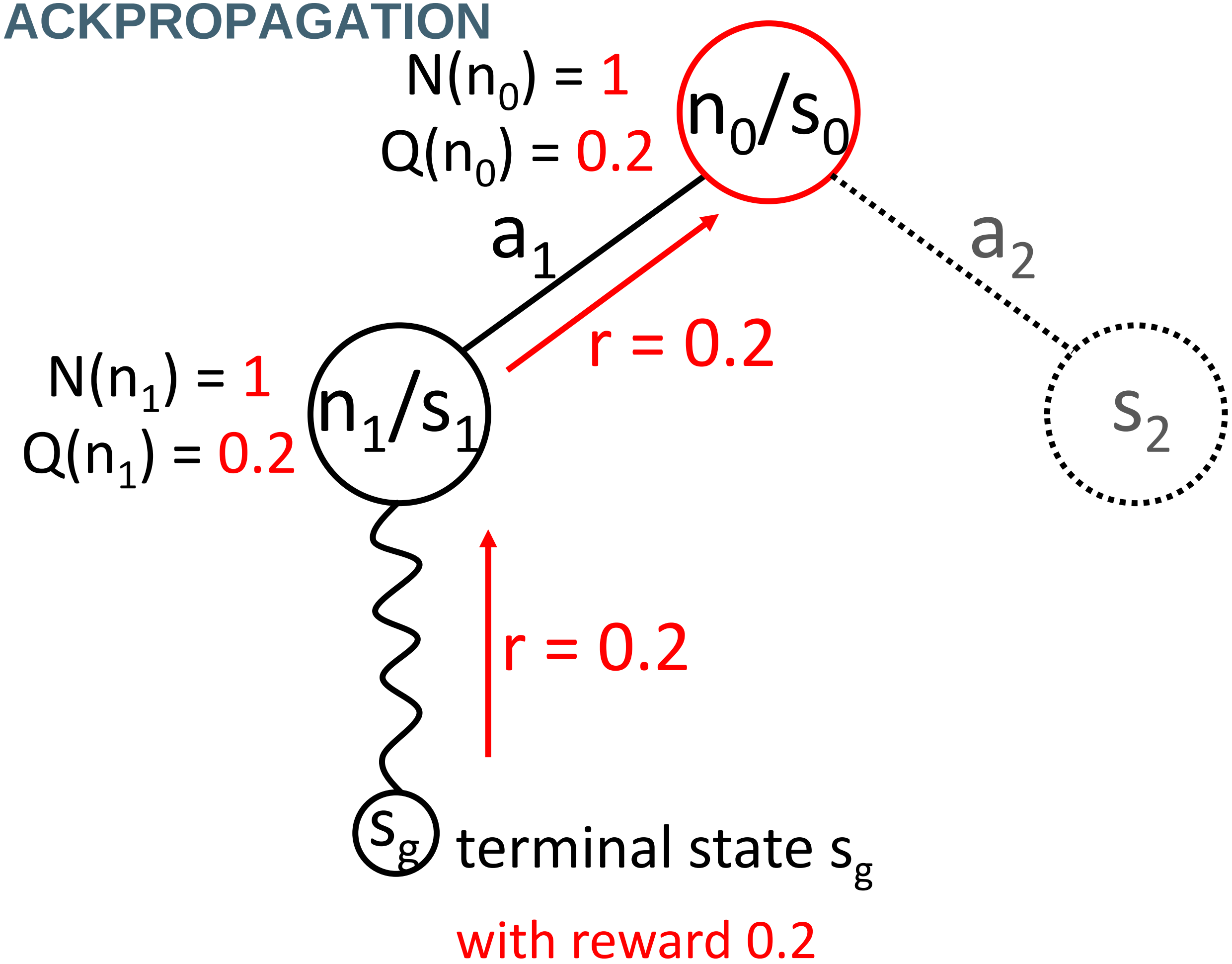
Run a simulation from the selected, unexplored node

(4) BACKPROPAGATION



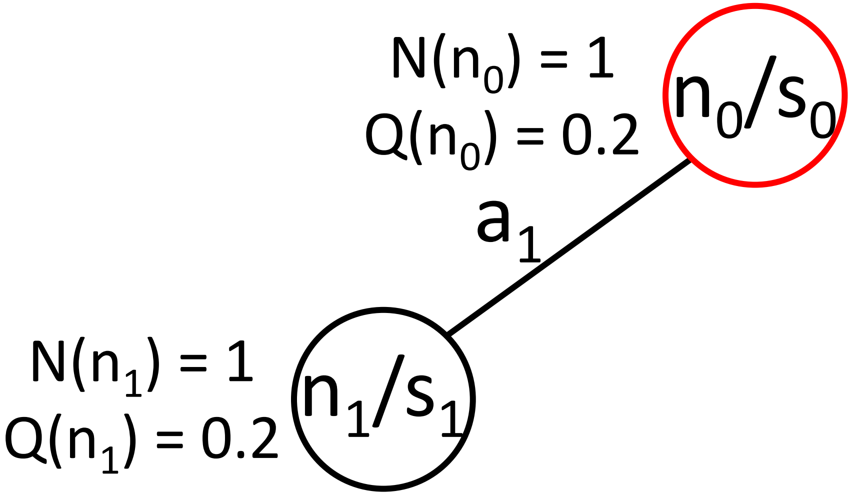
Backpropagate the reward up the path (only to nodes in the tree)

(4) Backpropagation

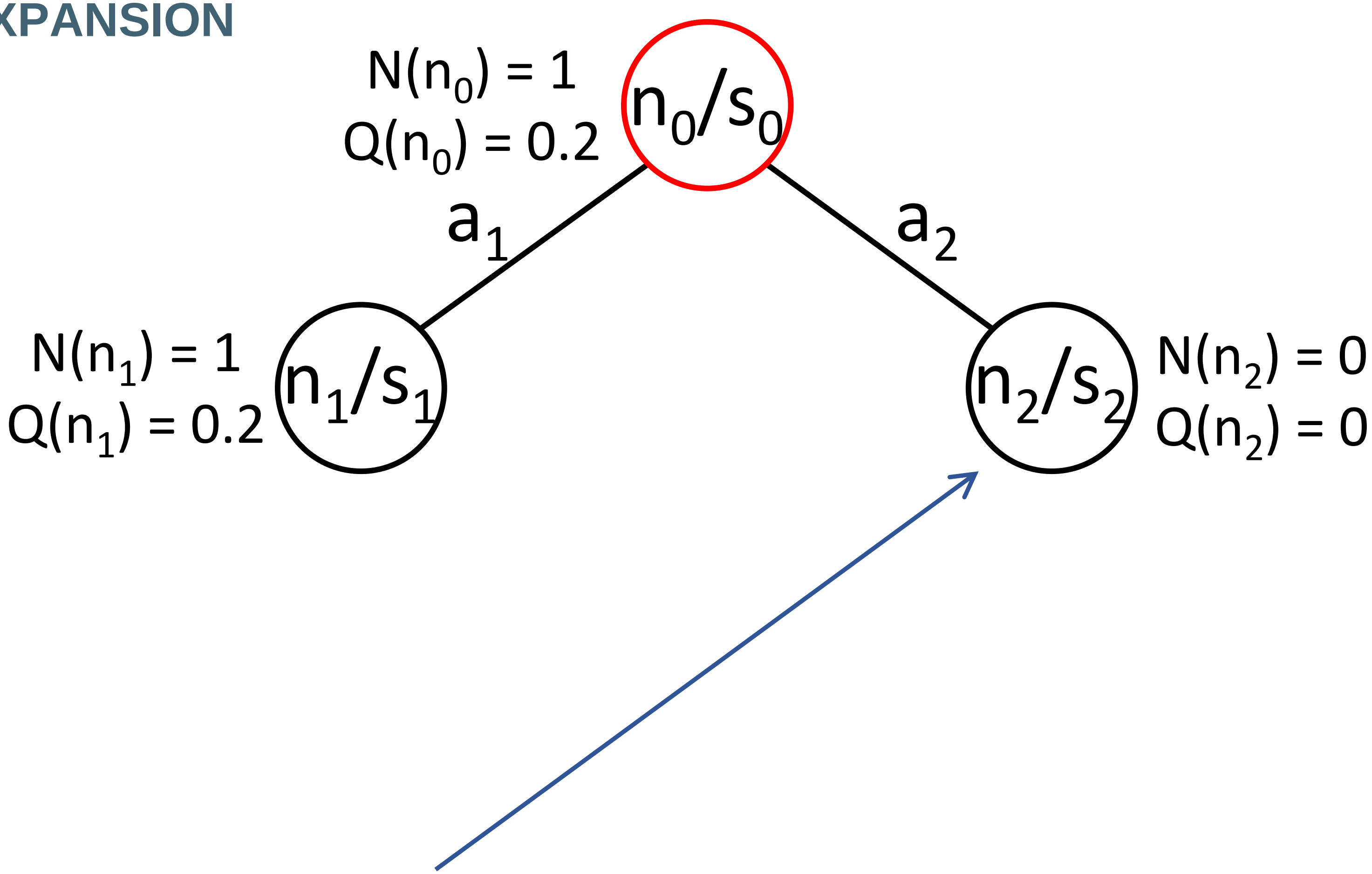


Backpropagate the reward up the path (only to nodes in the tree)

Search Tree after First Round

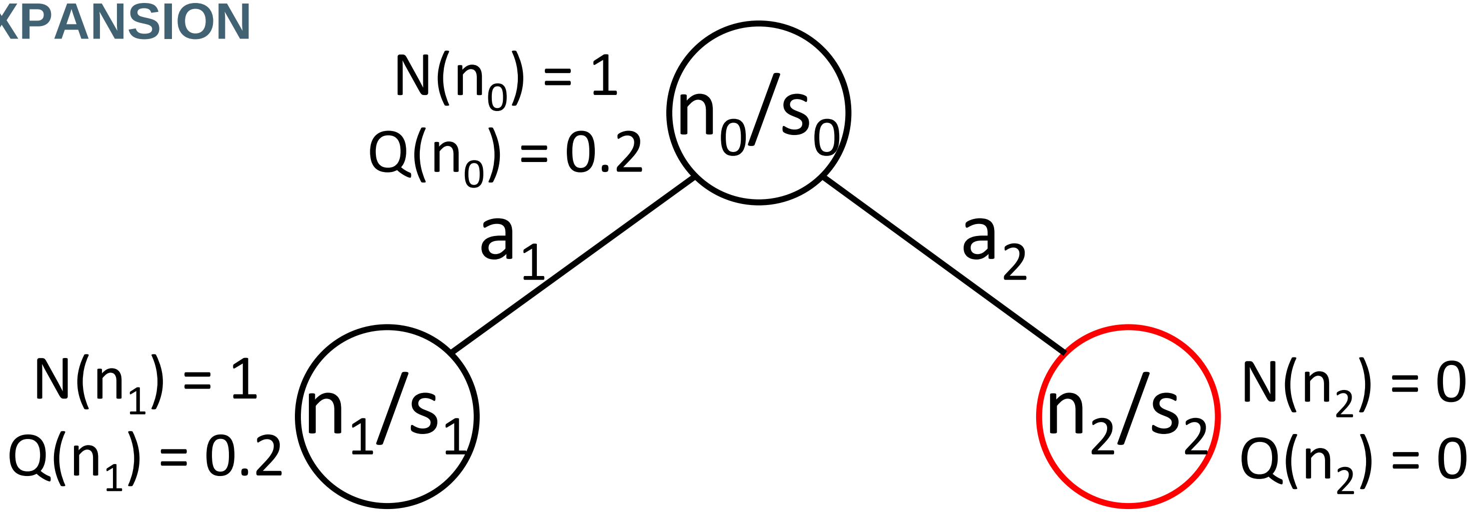


(2) Expansion

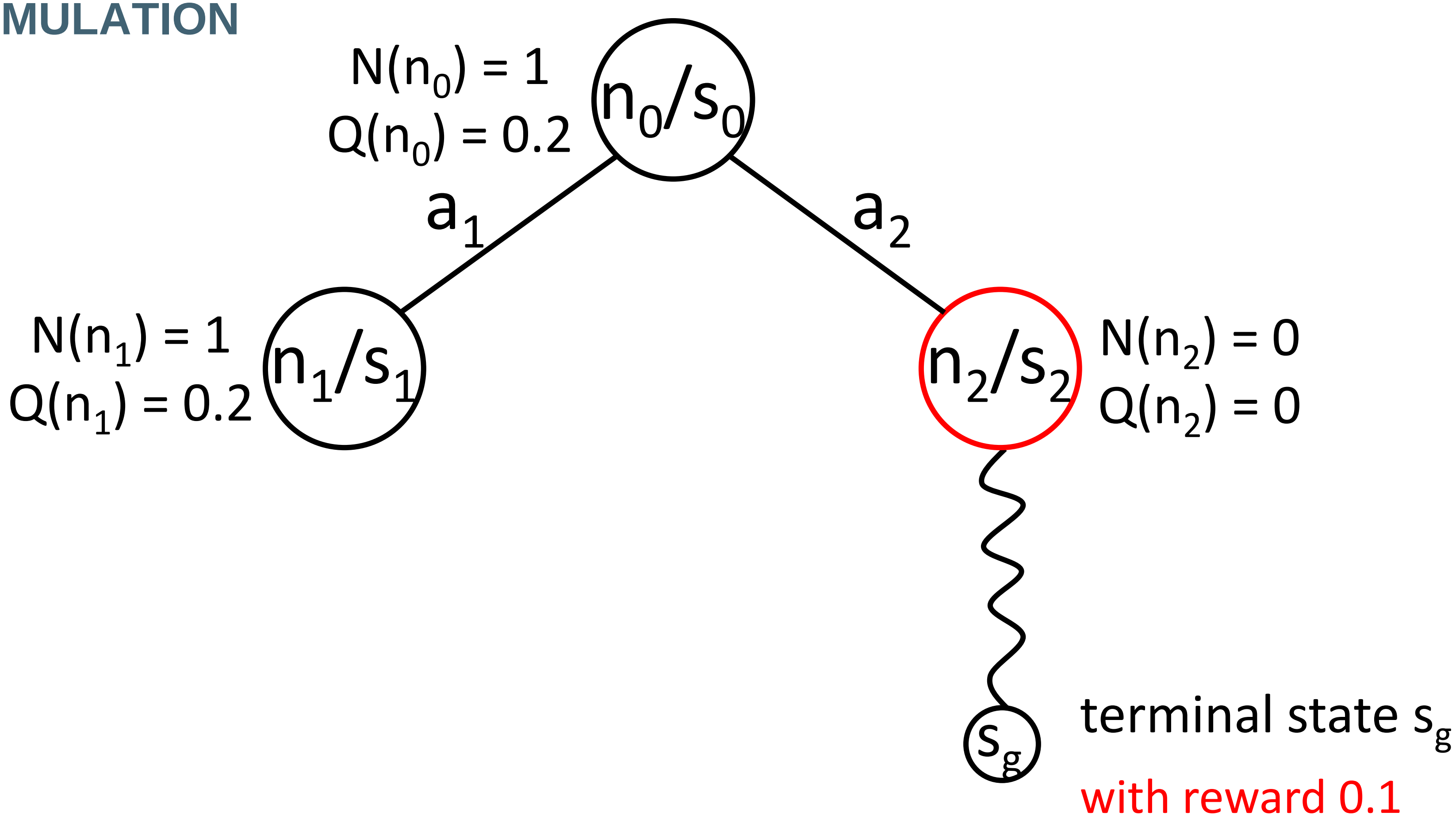


Expand unexplored child of selected node

(2) EXPANSION

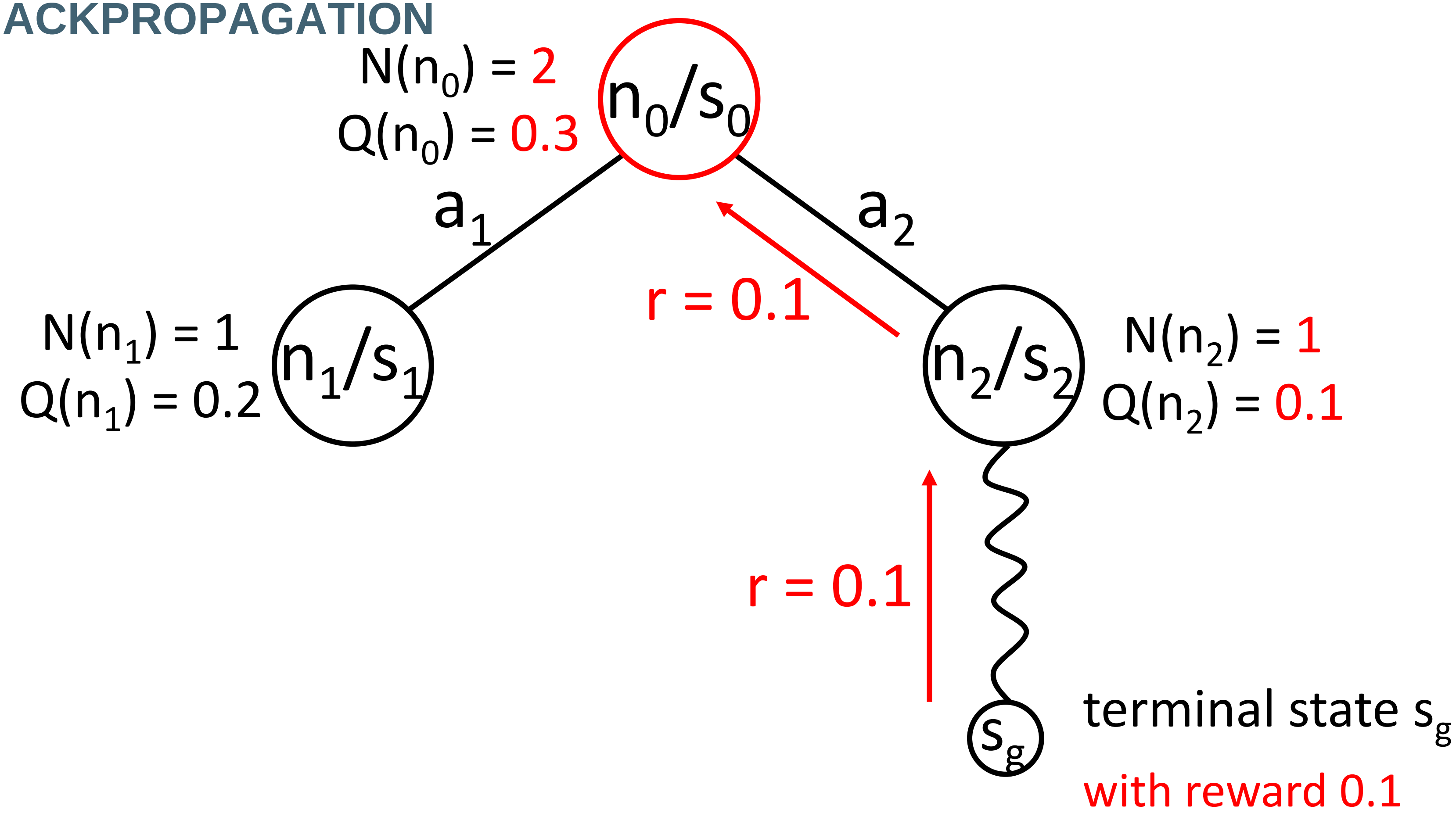


(3) SIMULATION



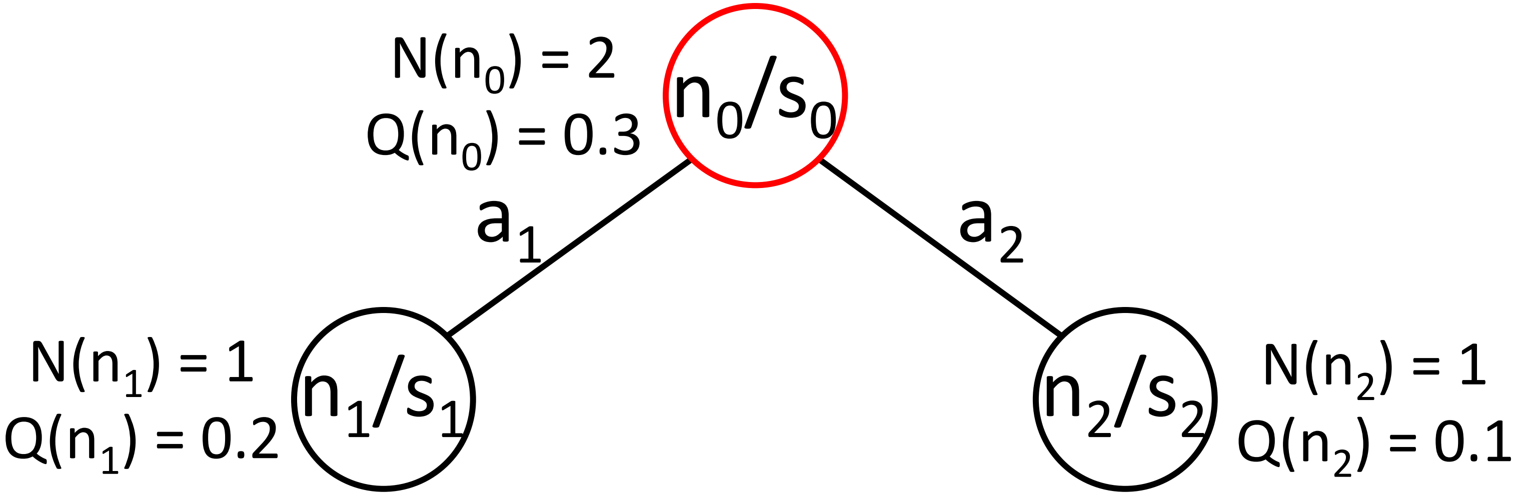
Run a simulation from the unexplored node

(4) BACKPROPAGATION



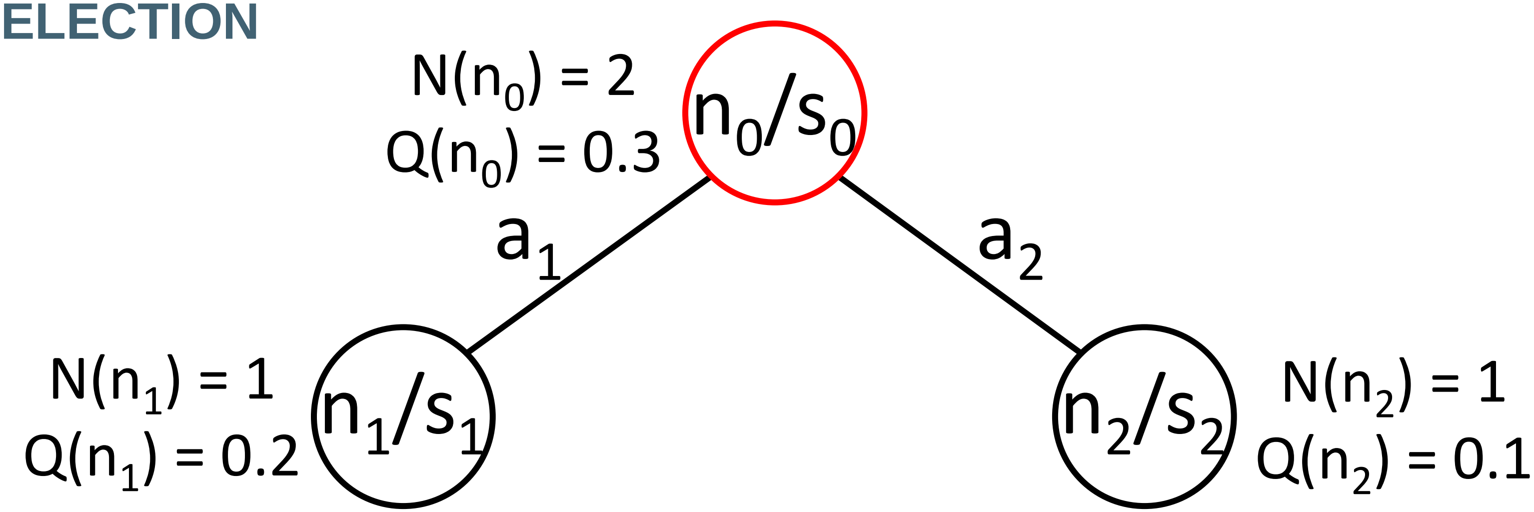
Backpropagate the reward value up the path

State after the Second Round



Now proceed with 3rd round …

(1) Selection



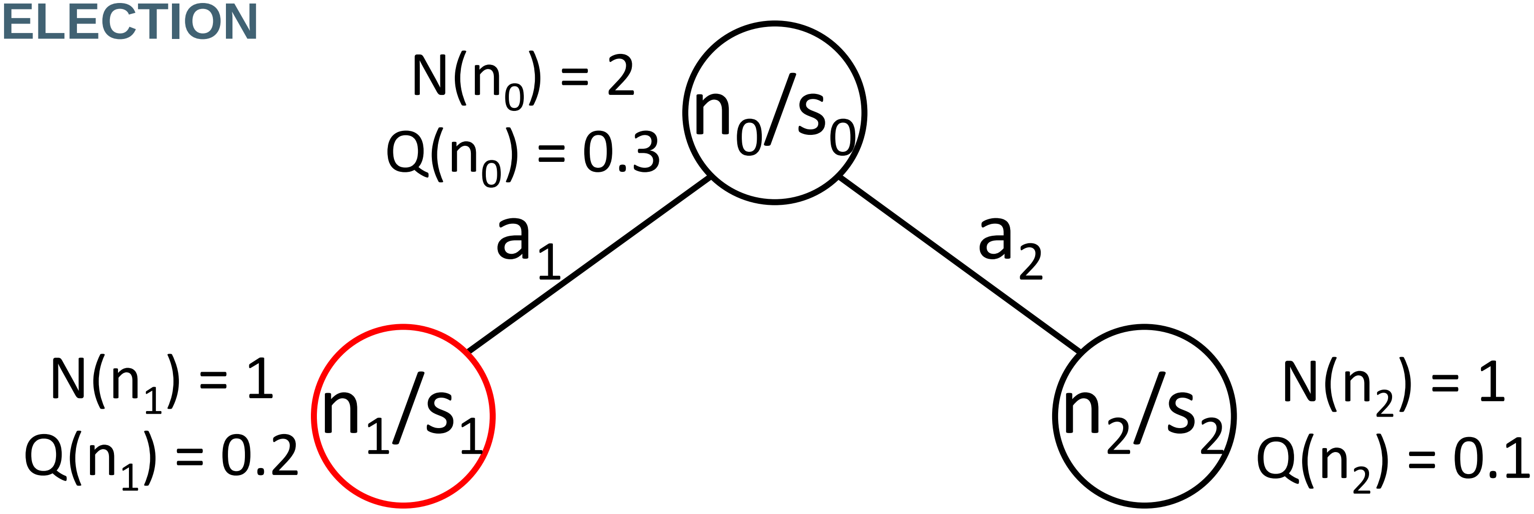
$$\text{UCB1}(n_1) = \frac{0.2}{1} + \sqrt{2}\sqrt{\frac{2\ln(2)}{1}} \approx 1.87 \qquad \text{UCB1}(n_2) = \frac{0.1}{1} + \sqrt{2}\sqrt{\frac{2\ln(2)}{1}} \approx 1.77$$

$$\boxed{\text{UCB1}(n) = \frac{Q(n)}{N(n)} + c \cdot \sqrt{\frac{2\ln(N)}{N(n)}} = \frac{Q(n)}{N(n)} + \sqrt{2} \cdot \sqrt{\frac{2\ln(N)}{N(n)}} = \frac{Q(n)}{N(n)} + 2\sqrt{\frac{\ln(N)}{N(n)}}}$$

Where we have chosen $c = \sqrt{2}$

Calculate the UCB1 values to decide which node to select

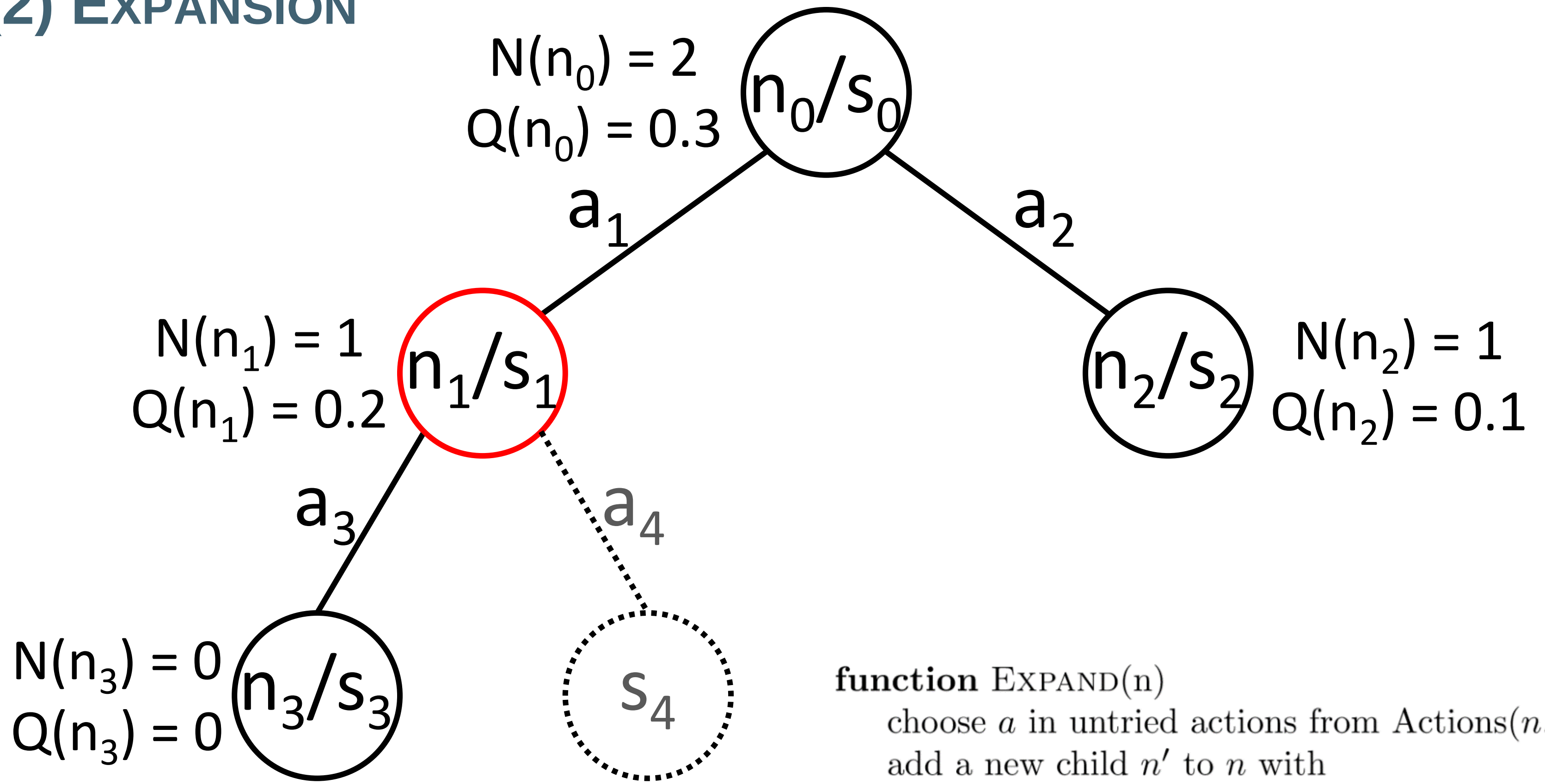
(1) SELECTION



$$\text{UCB1}(n_1) = \frac{0.2}{1} + 2\sqrt{\frac{\ln(2)}{1}} \approx 1.87 \qquad \text{UCB1}(n_2) = \frac{0.1}{1} + 2\sqrt{\frac{\ln(2)}{1}} \approx 1.77$$

Select the child with highest UCB1 value

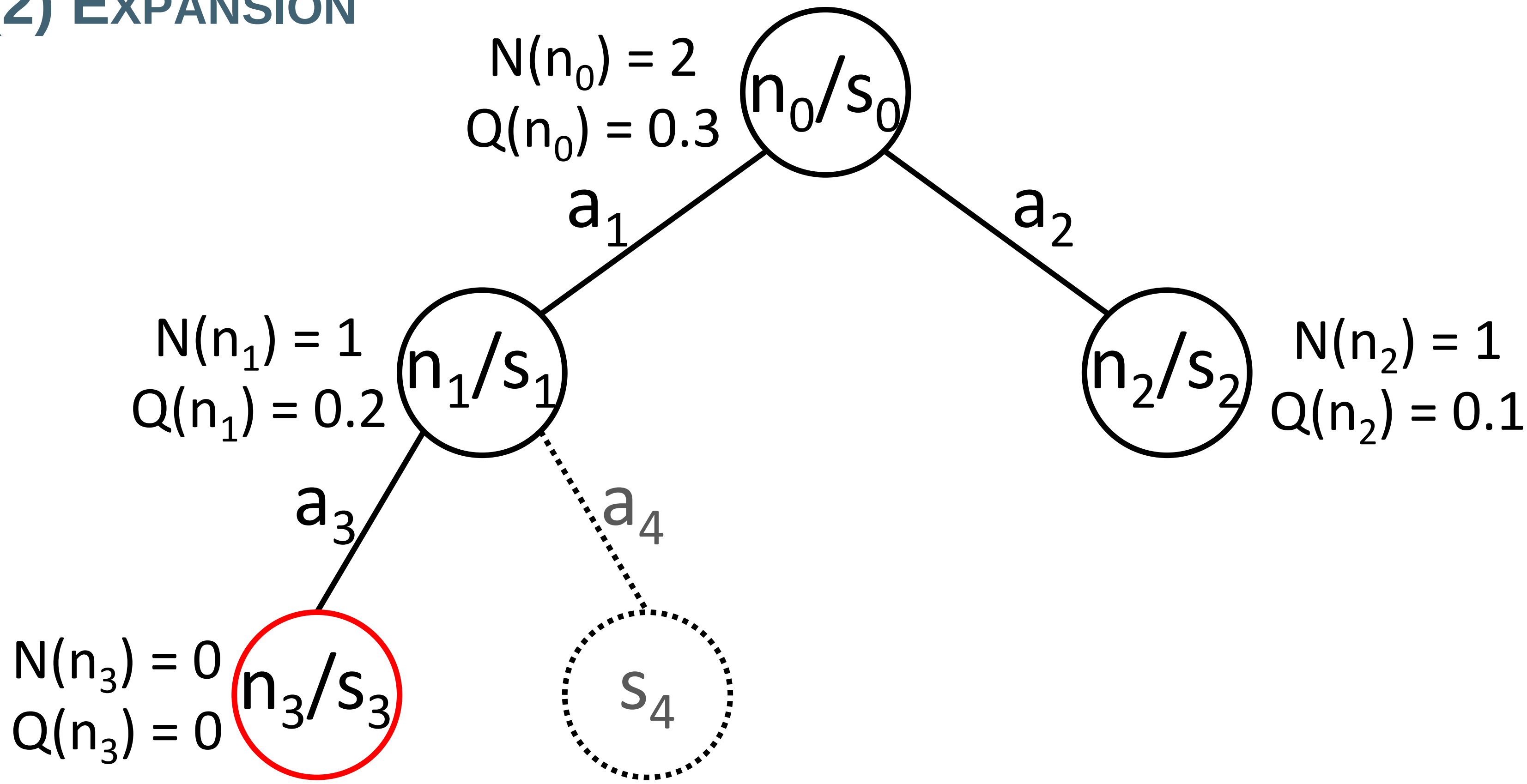
(2) Expansion



function EXPAND(n)
  choose a in untried actions from Actions(n.State)
  add a new child n' to n with
  n'.State = ChildState(n.State, a)
  $N(n') = 0$
  $Q(n') = 0$
  return n'

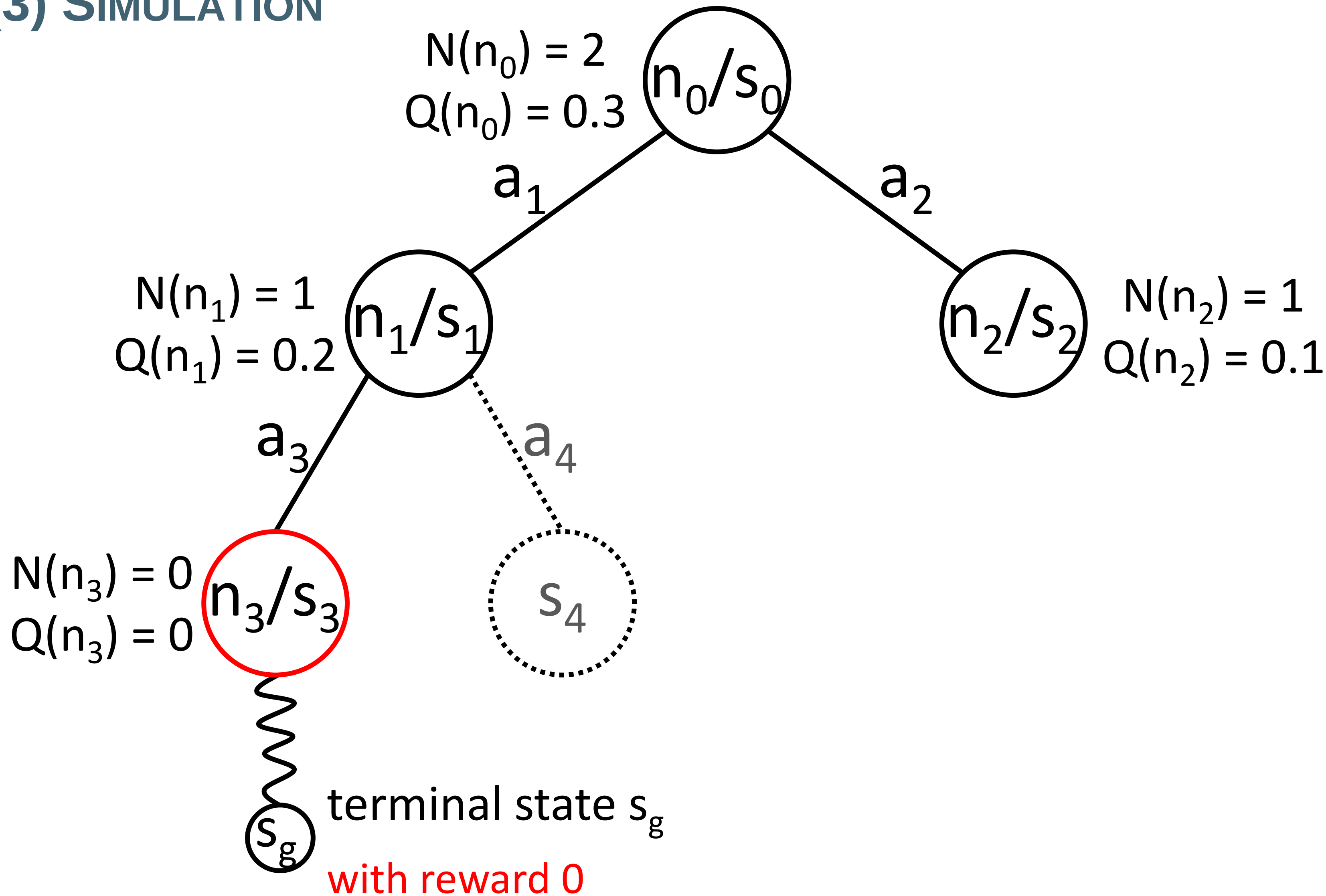
Expand the selected node

(2) EXPANSION



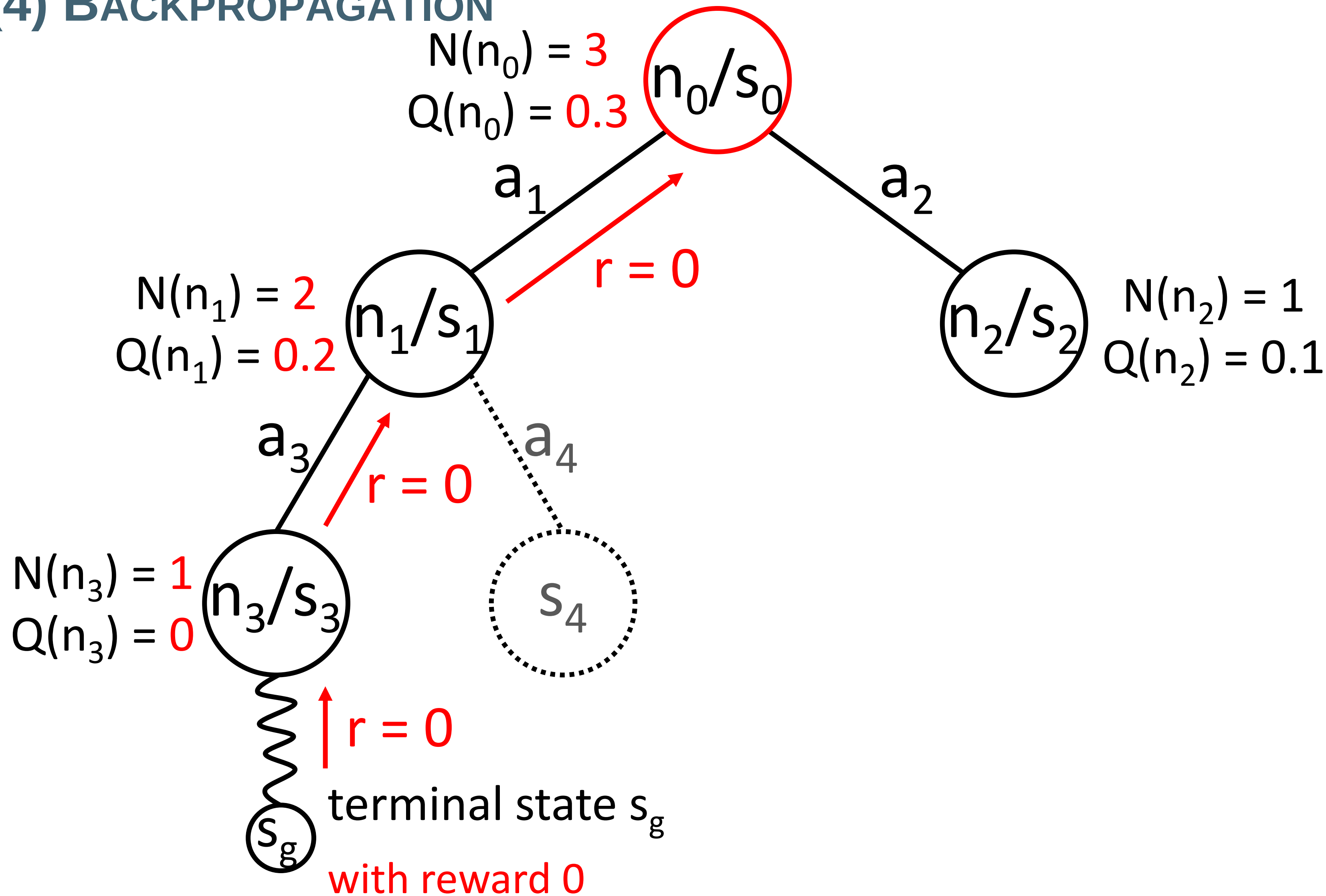
$N(n_0) = 2$
$Q(n_0) = 0.3$
n_0/s_0
a_1
a_2
$N(n_1) = 1$
$Q(n_1) = 0.2$
n_1/s_1
n_2/s_2
$N(n_2) = 1$
$Q(n_2) = 0.1$
a_3
a_4
$N(n_3) = 0$
$Q(n_3) = 0$
n_3/s_3
s_4

(3) SIMULATION



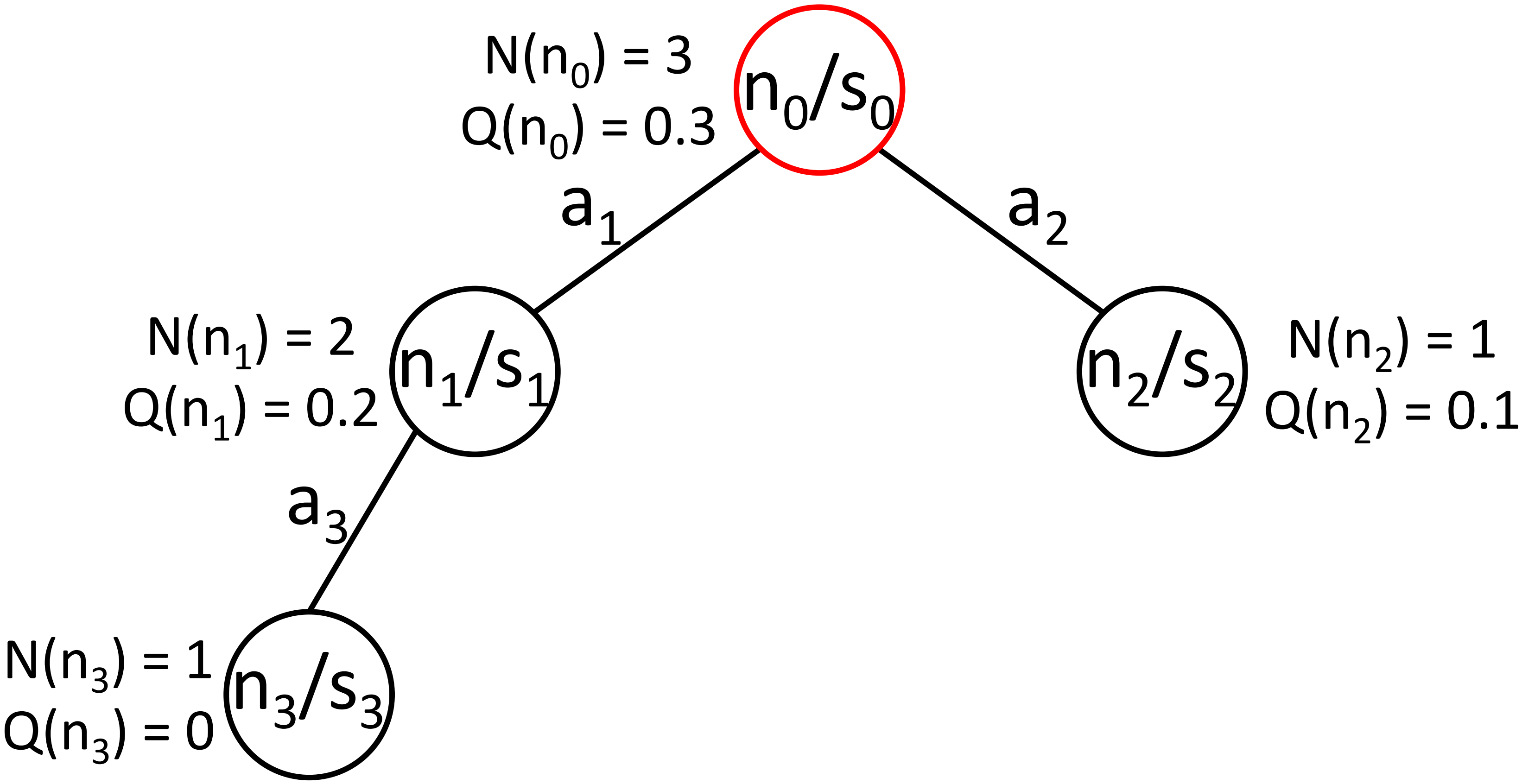
Run a simulation from the unexplored node

(4) Backpropagation

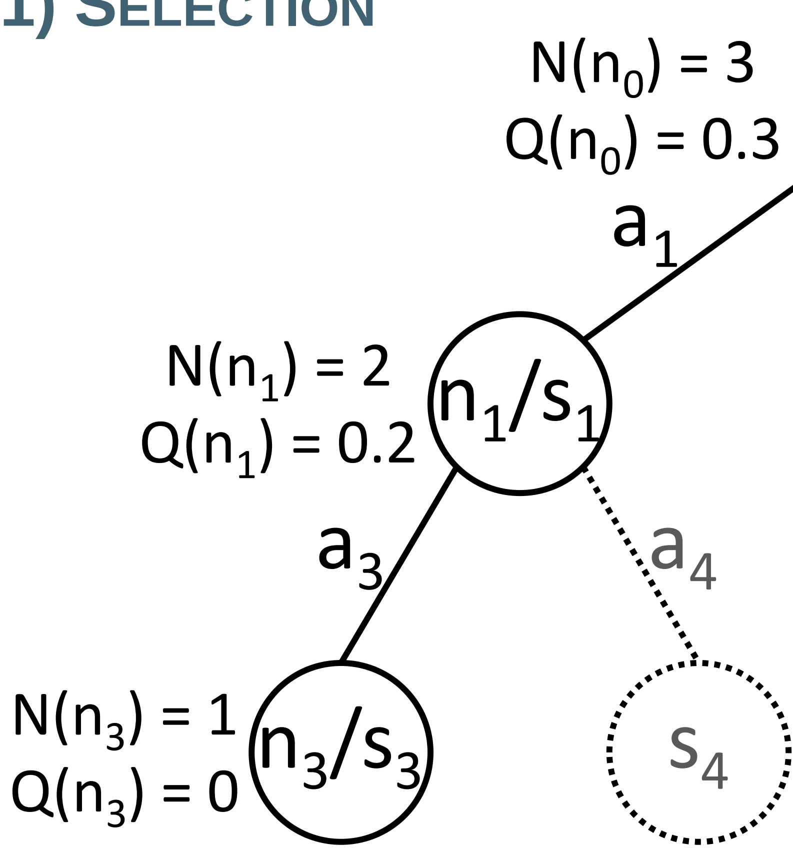


Backpropagate the reward up the path

State after the Third Round



(1) Selection

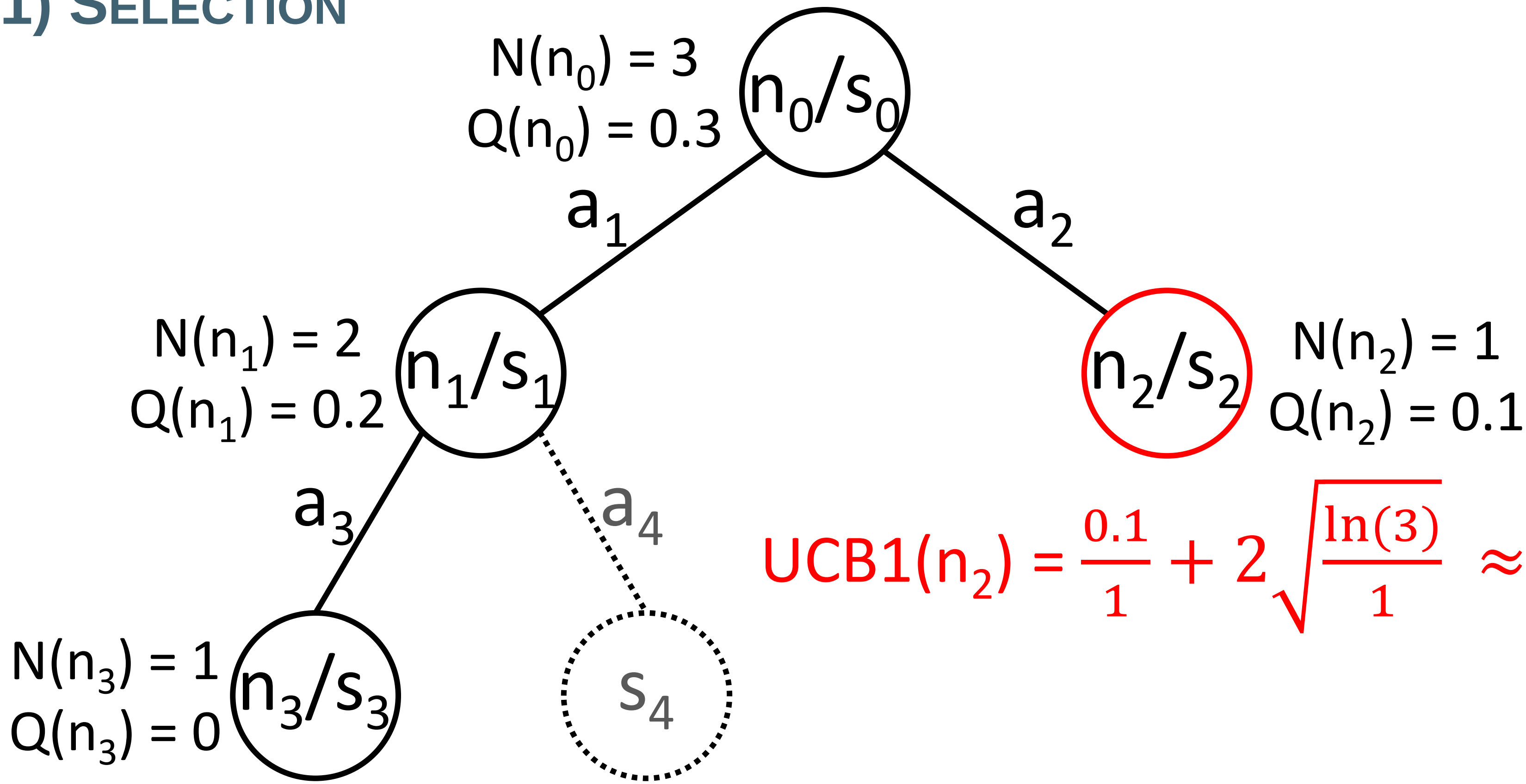


$$\text{UCB1}(n_2) = \frac{0.1}{1} + 2\sqrt{\frac{\ln(3)}{1}} \approx 2.20$$

$$\text{UCB1}(n_1) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(3)}{2}} \approx 1.58$$

Calculate the UCB1 values

(1) SELECTION

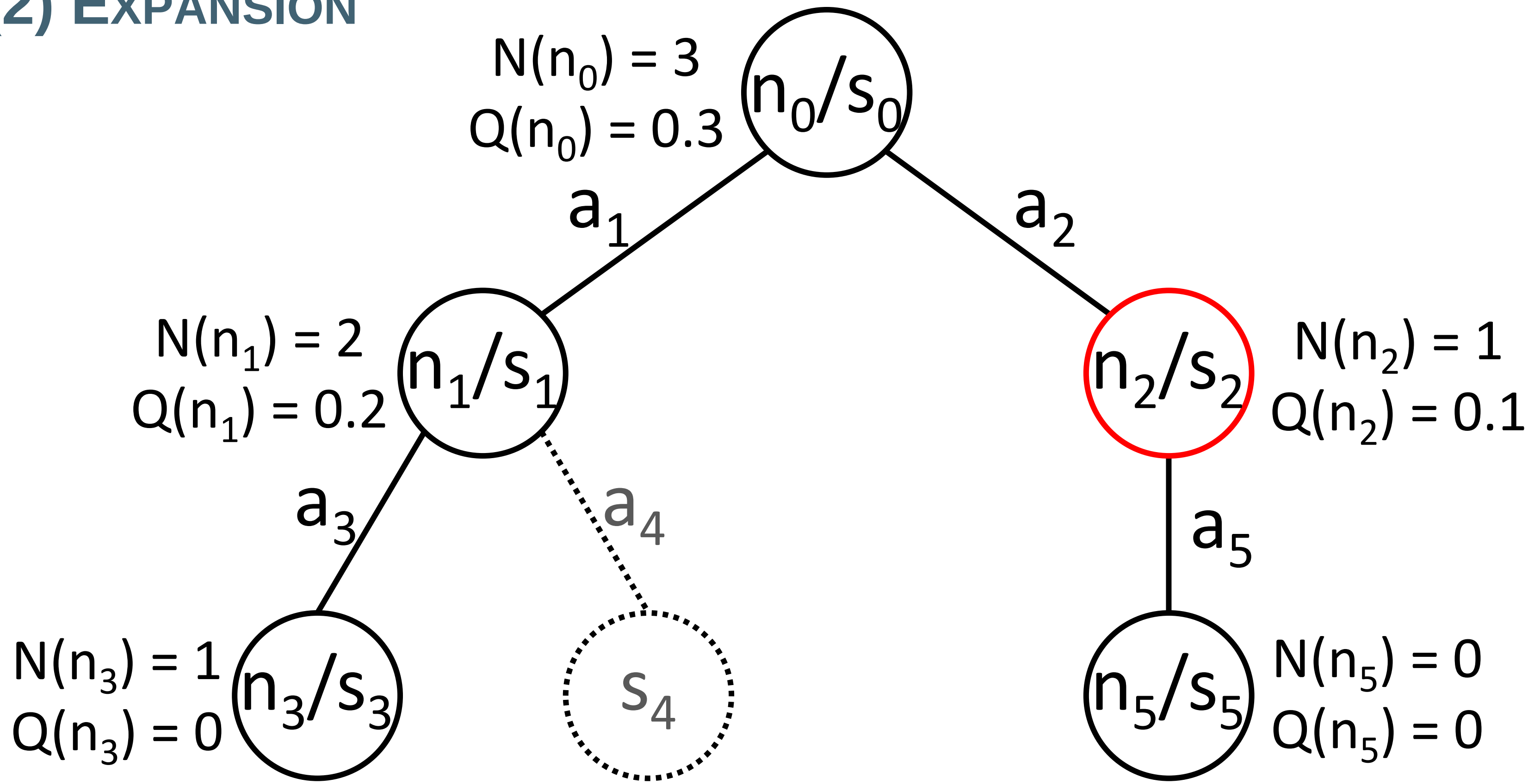


$$\mathrm{UCB1}(n_2) = \frac{0.1}{1} + 2\sqrt{\frac{\ln(3)}{1}} \approx 2.20$$

$$\mathrm{UCB1}(n_1) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(3)}{2}} \approx 1.58$$

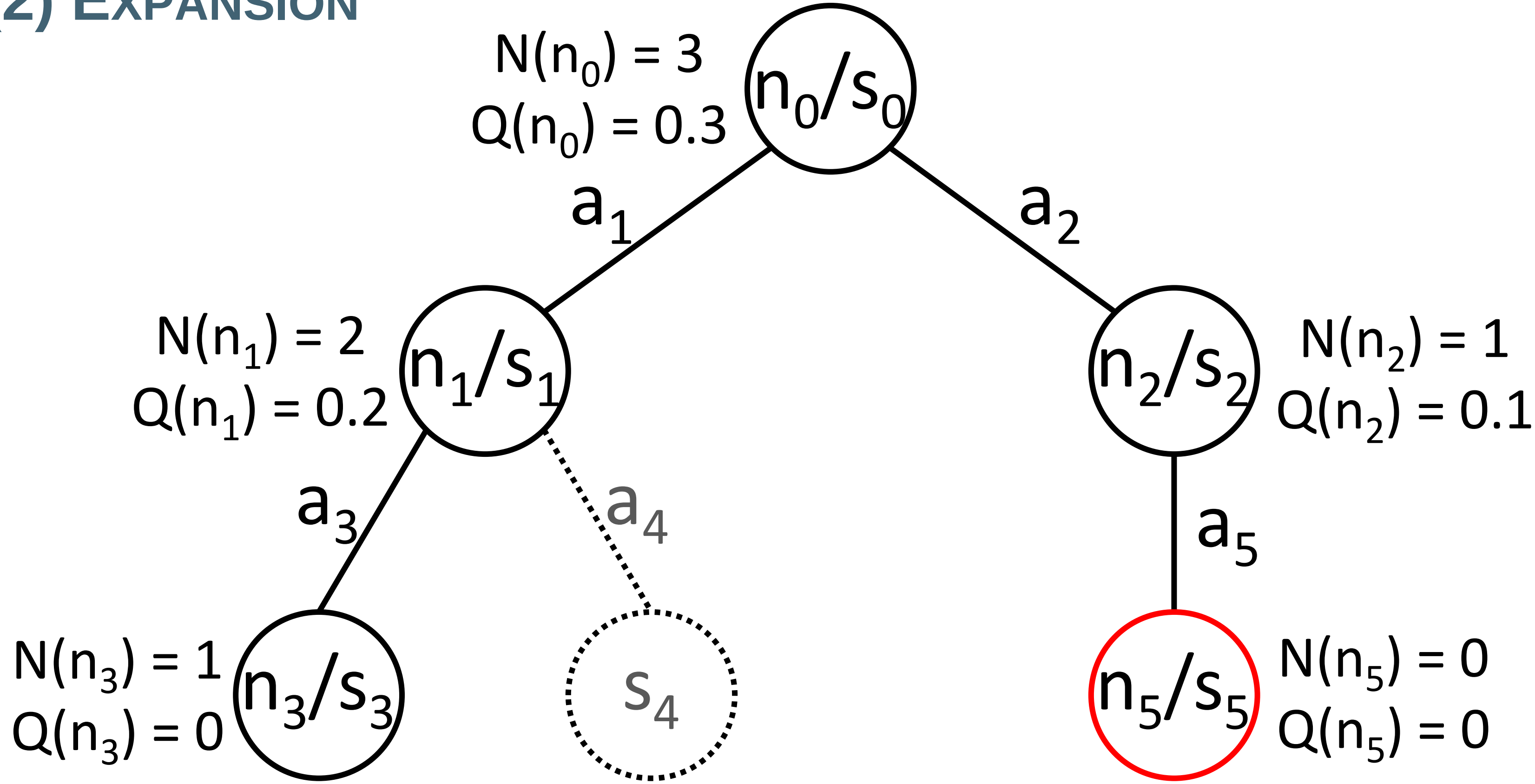
Select the child with highest UCB1 value

(2) EXPANSION

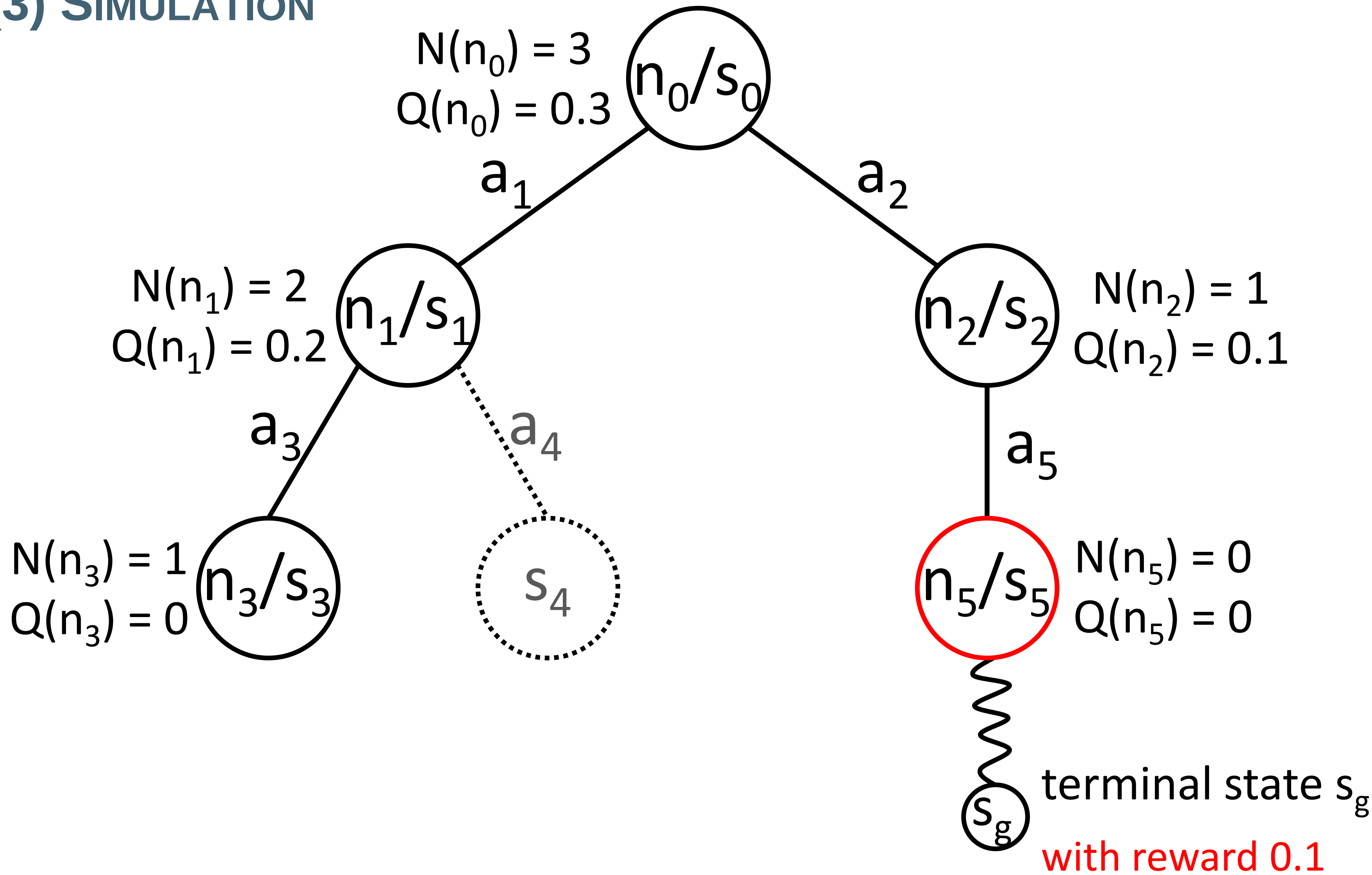


Expand selected node

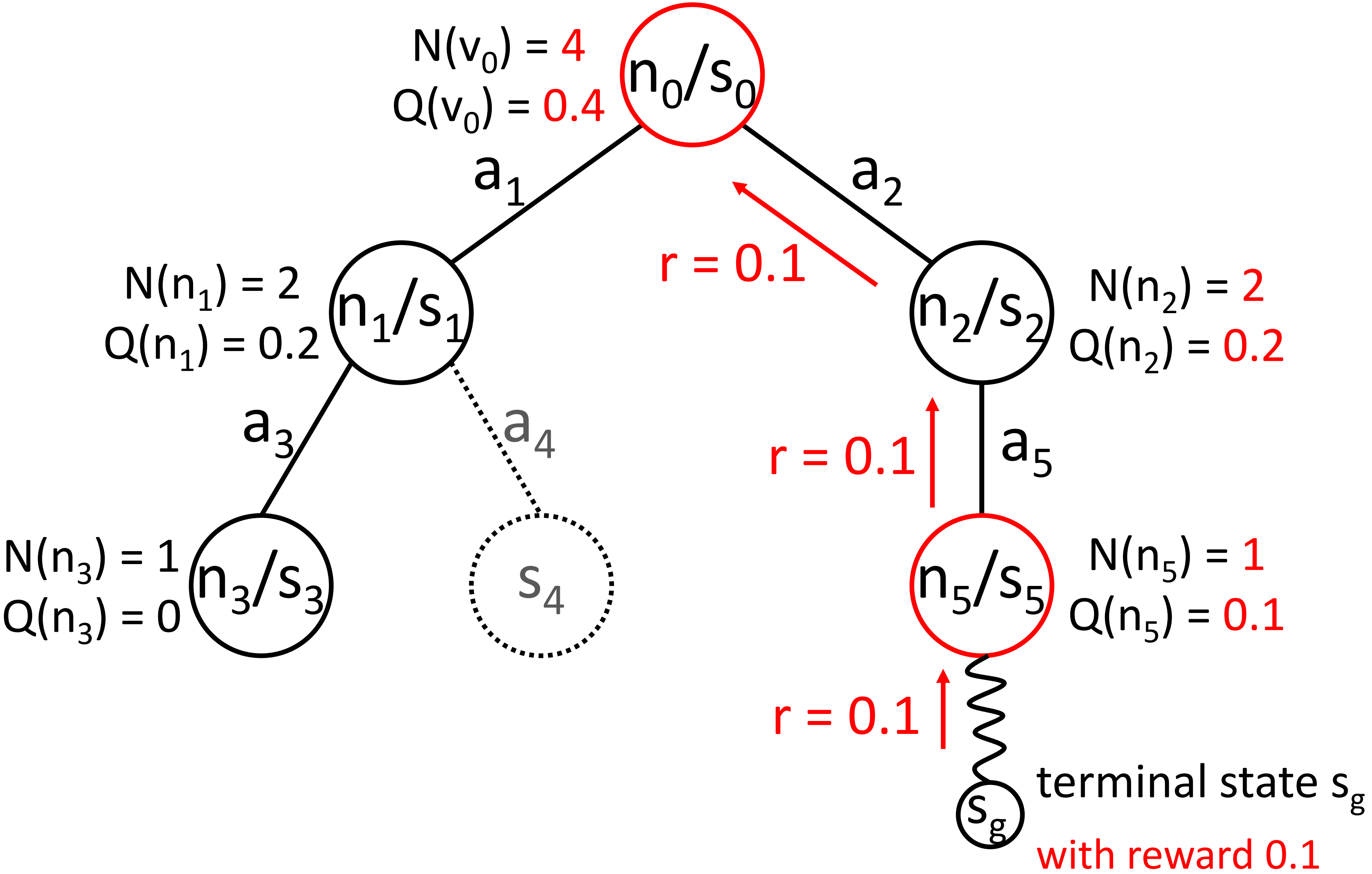
(2) EXPANSION


N(n0) = 3
Q(n0) = 0.3
n0/s0
a1
a2
N(n1) = 2
Q(n1) = 0.2
n1/s1
n2/s2
N(n2) = 1
Q(n2) = 0.1
a3
a4
a5
N(n3) = 1
Q(n3) = 0
n3/s3
s4
n5/s5
N(n5) = 0
Q(n5) = 0

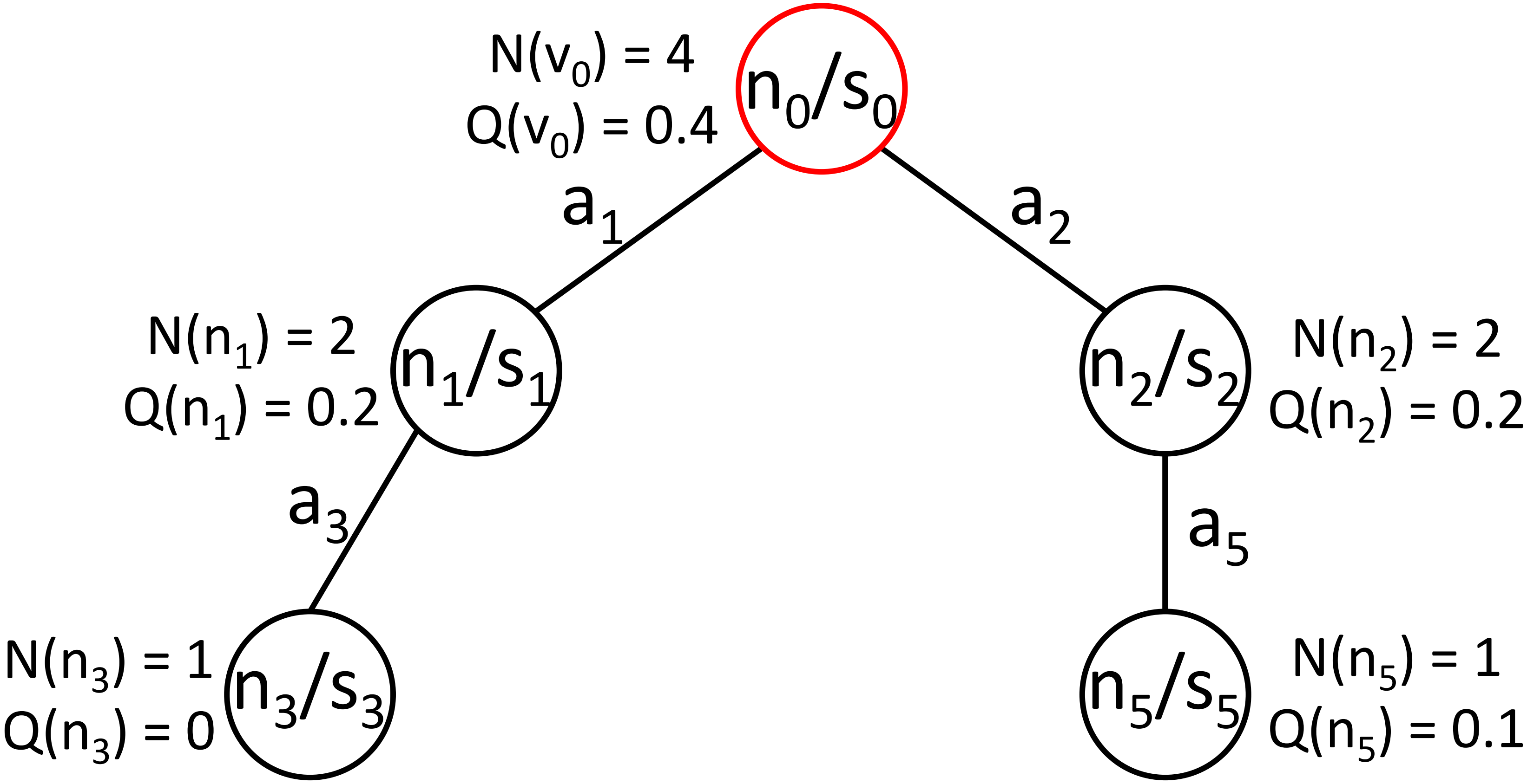
(3) SIMULATION



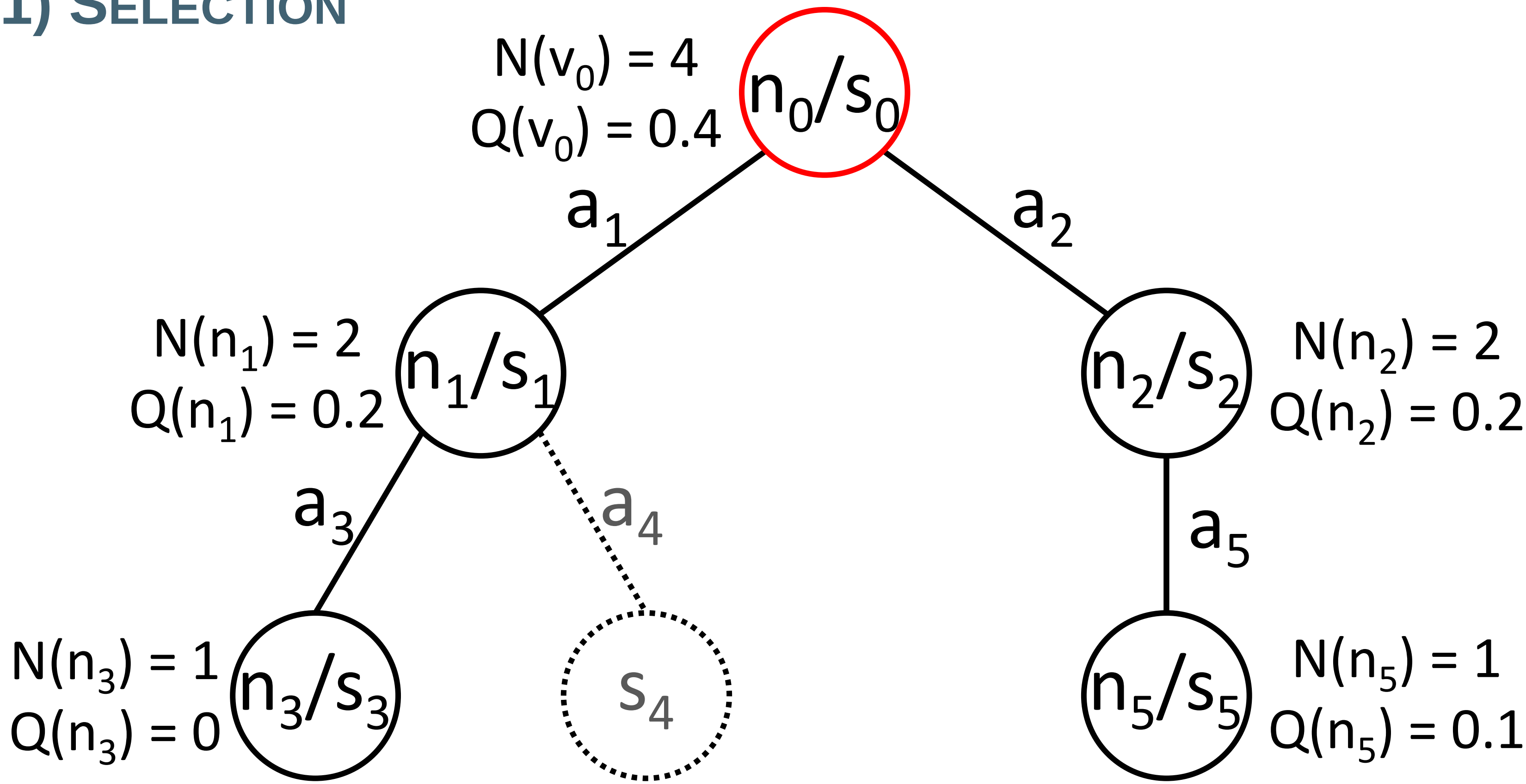
(4) BACKPROPAGATION



State after Fourth Round



(1) Selection

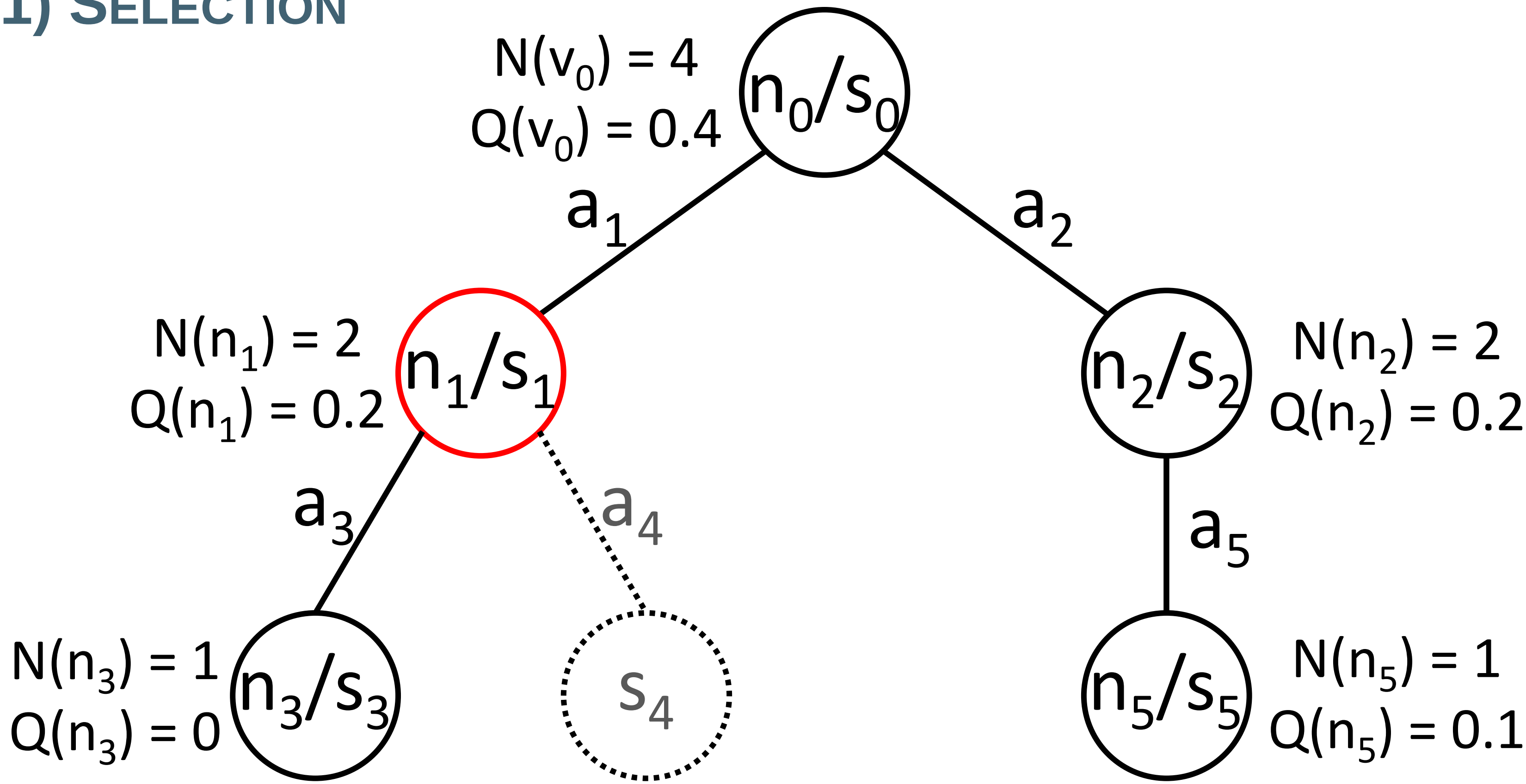


$\mathrm{UCB1}(n_1) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(4)}{2}} \approx 1.77$

$\mathrm{UCB1}(n_2) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(4)}{2}} \approx 1.77$

Calculate UCB1 values

(1) Selection

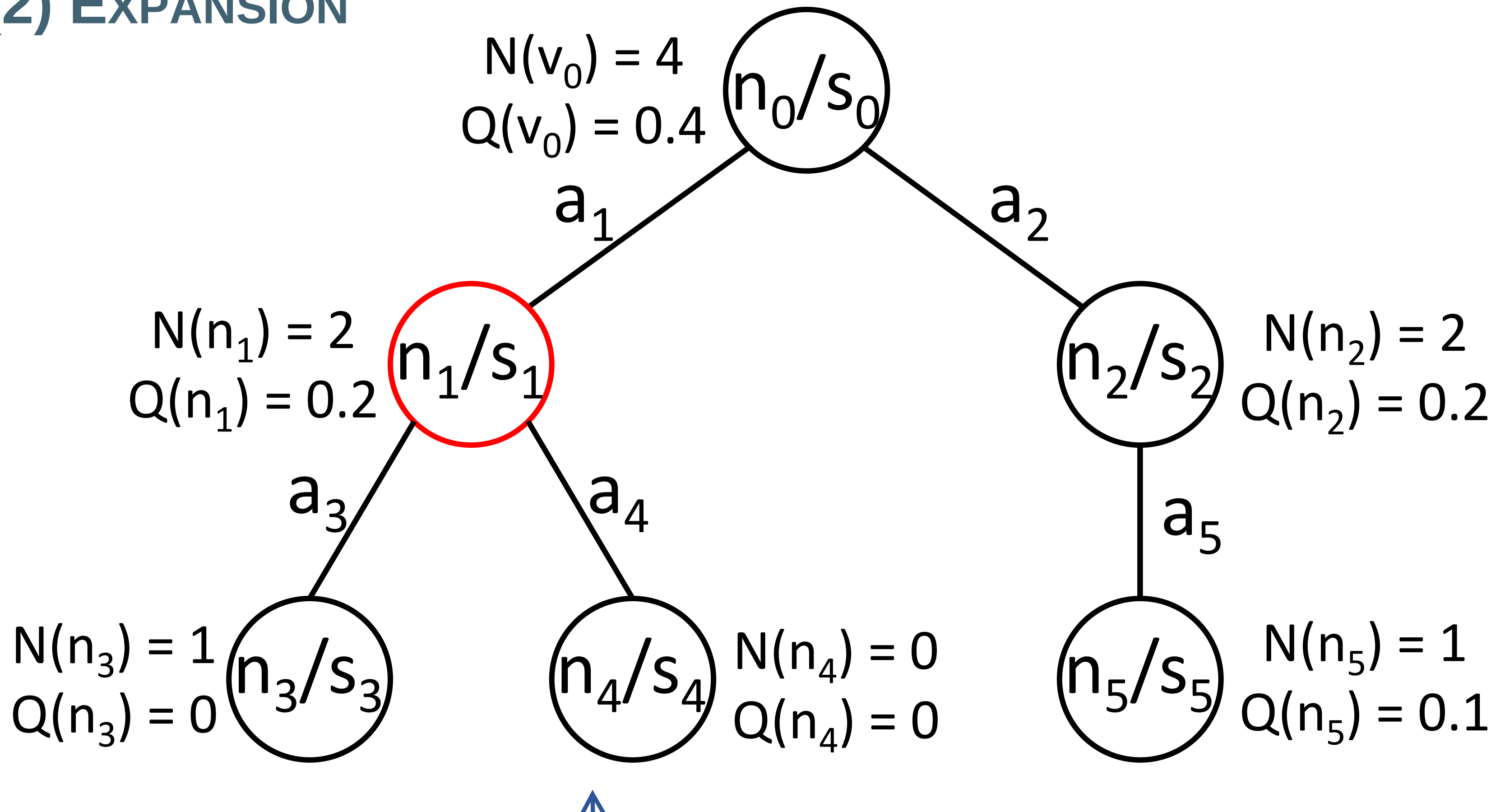


$UCB1(n_1) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(4)}{2}} \approx 1.77$

$UCB1(n_2) = \frac{0.2}{2} + 2\sqrt{\frac{\ln(4)}{2}} \approx 1.77$

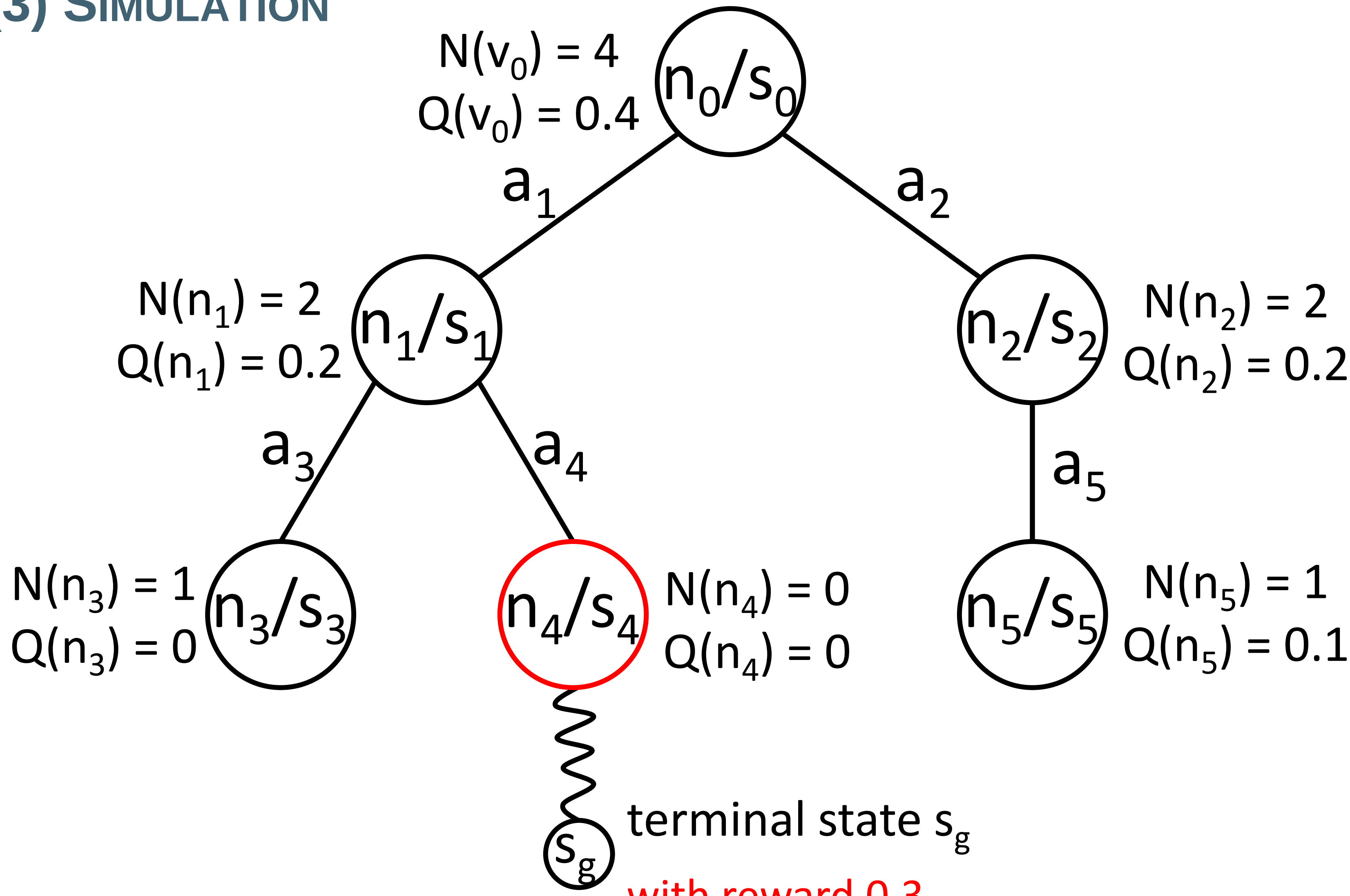
Break ties on identical UCB1 values (strategy: leftmost node first)

(2) EXPANSION

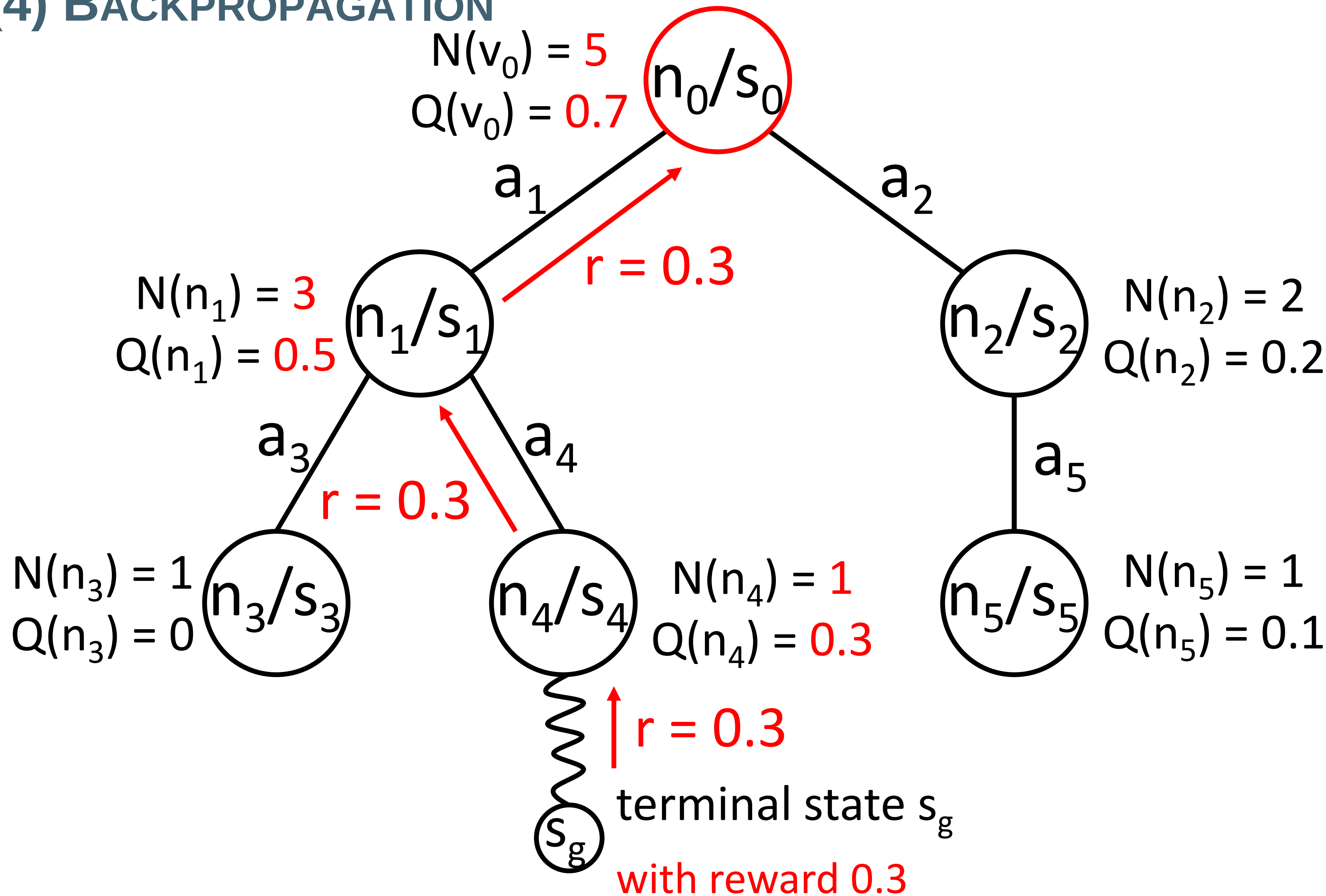


Expand selected node towards unexplored child n_4

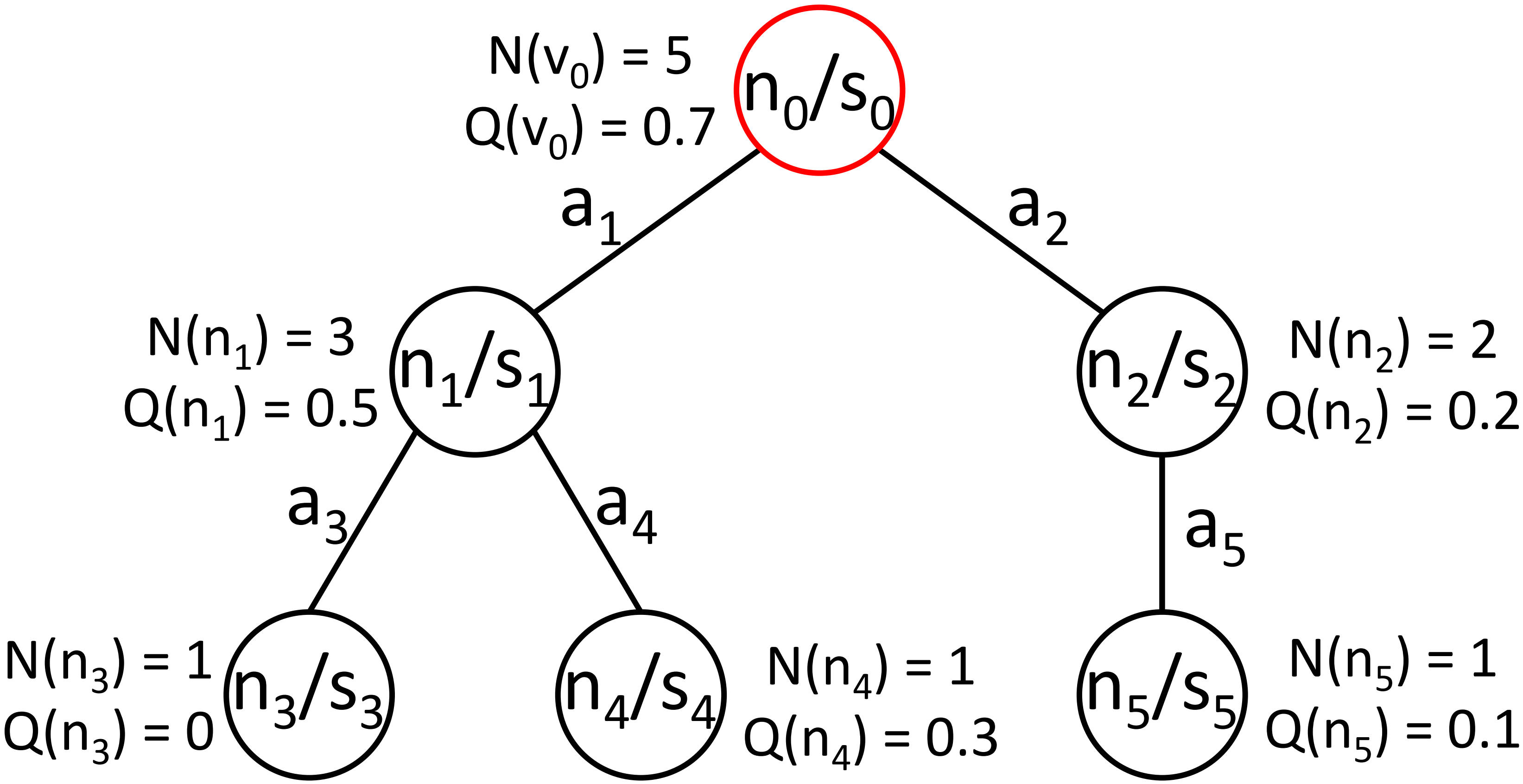
(3) SIMULATION



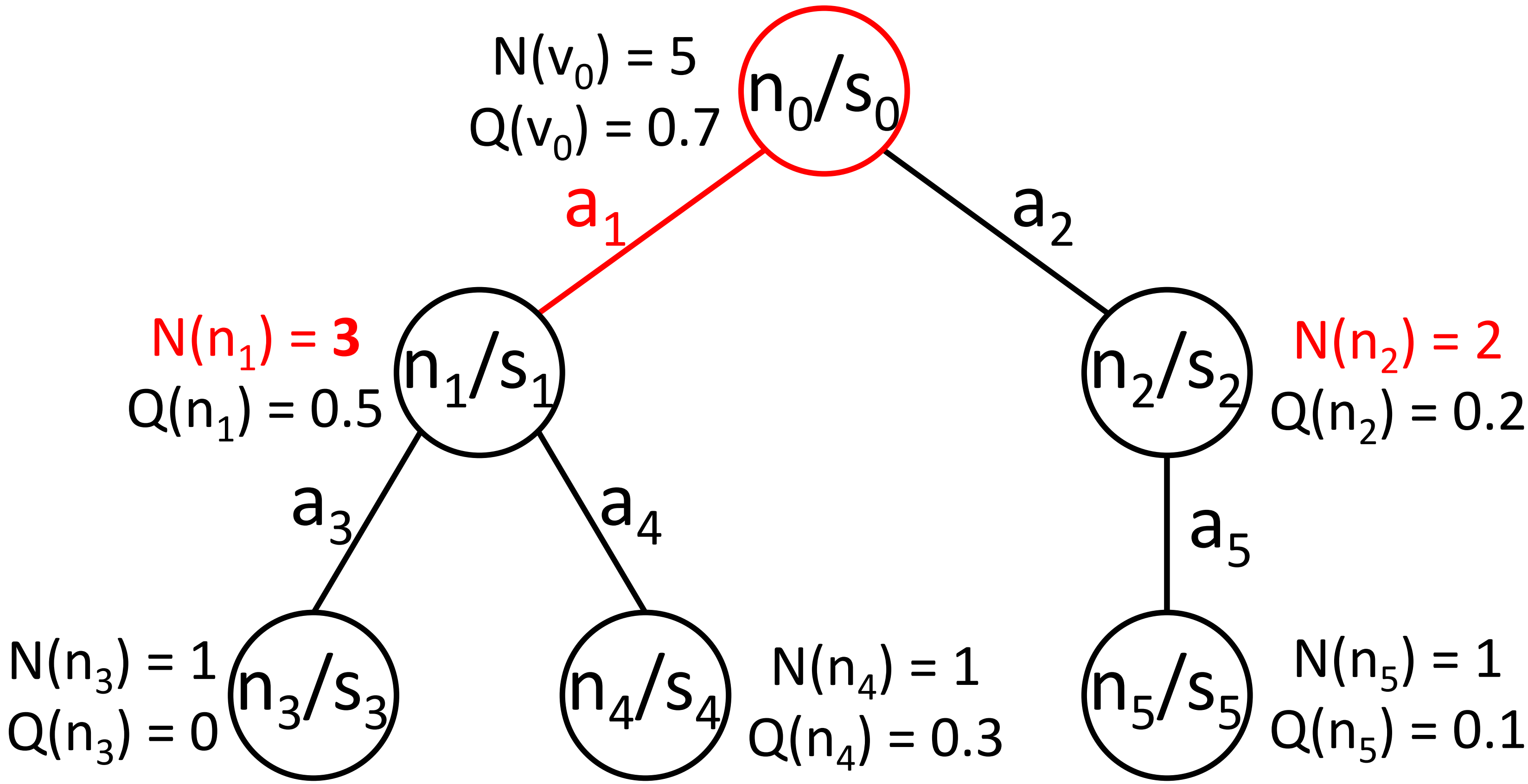
(4) BACKPROPAGATION



State after Fifth Round



Computational Budget Reached

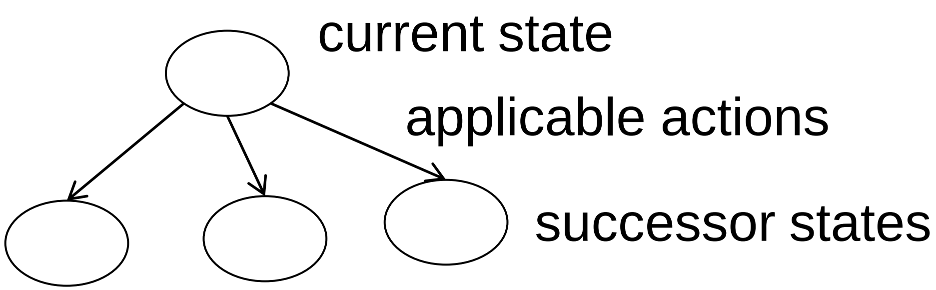


Depends on strategy, here: action leading to child node with most visits returned (most visited often means „best“)

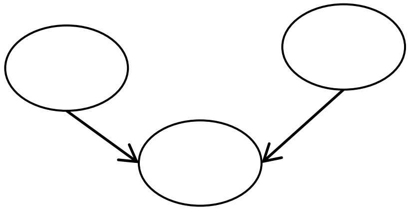
RETURN a_1

(4) Genetic Algorithms

- So far, all search strategies are based on expanding a single current state



- Why not take 2 parent states and combine it into a new successor state?

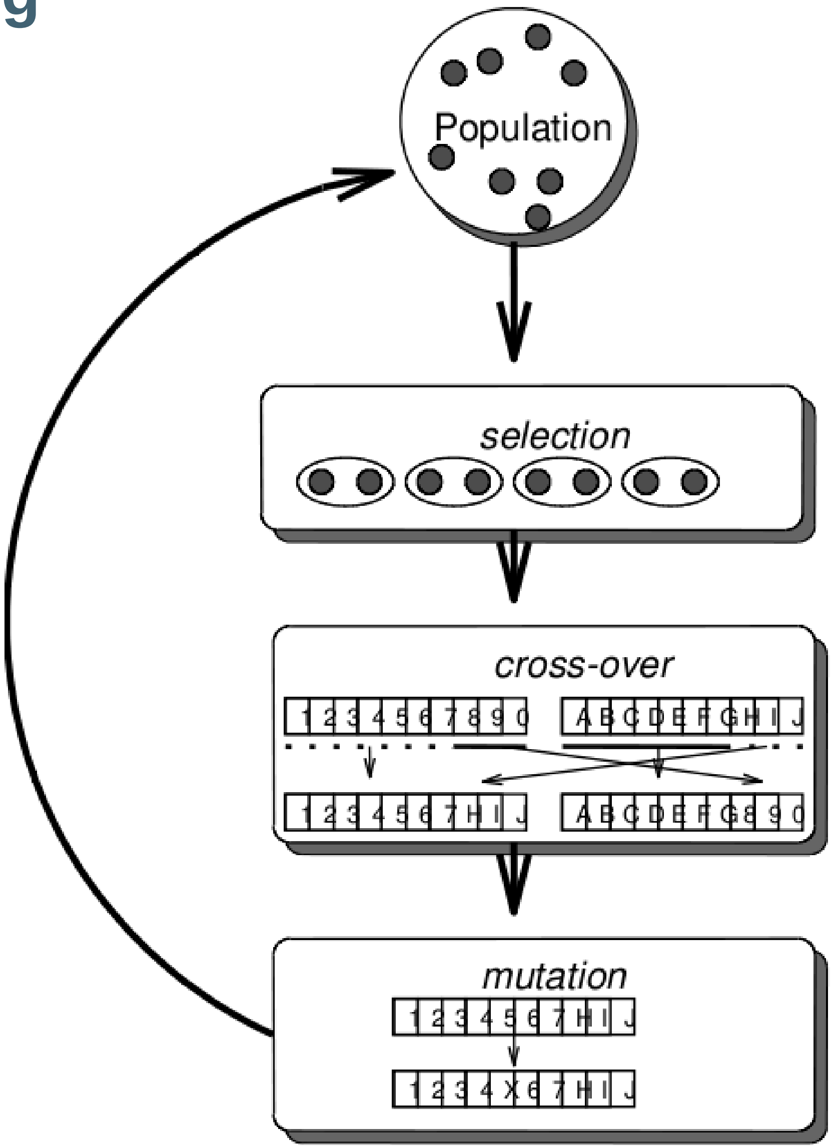


Basic Ideas in Genetic Algorithms

- Evolution seems to be good to produce good solutions
- Similar to evolution, search for solutions by sexual reproduction
  - combine 2 genoms by *crossing*, *mutating*, and *selecting*
- Ingredients
  - Encode a state as a string (gene)
  - Fitness function to evaluate states
  - Population of states (genes)
- https://www.youtube.com/watch?v=Y-XMh-iw07w

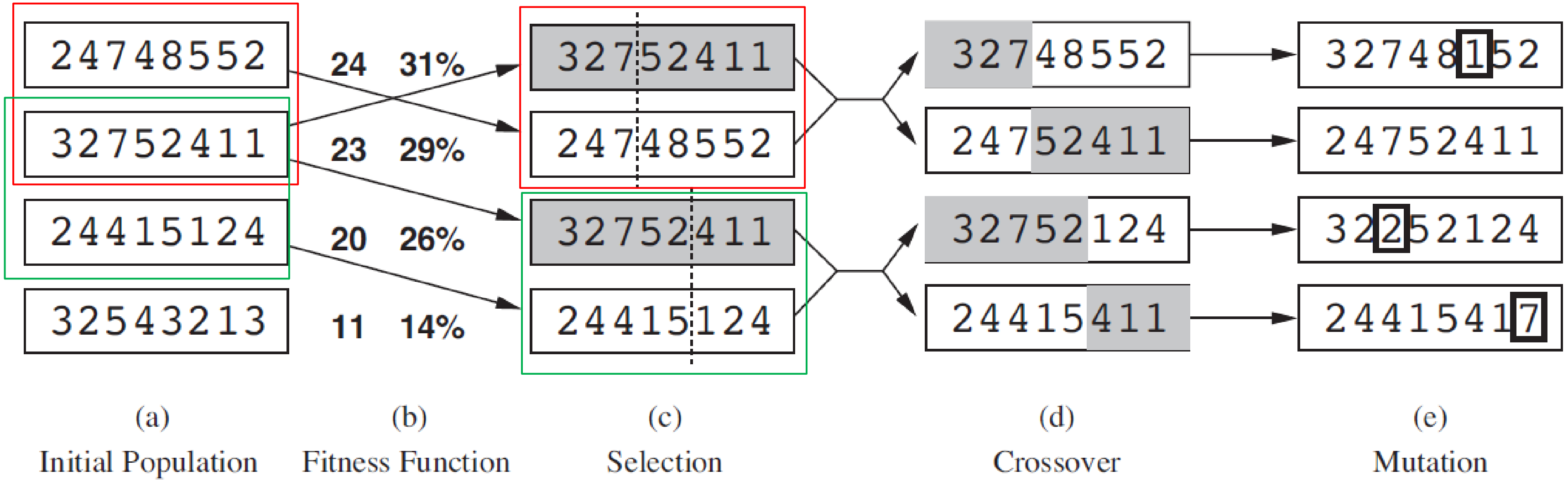
Selection, Mutation, and Crossing

- Many variation points, e.g.
  - how to select
  - what type of cross-over (e.g. where to break)
  - probability and type of mutations

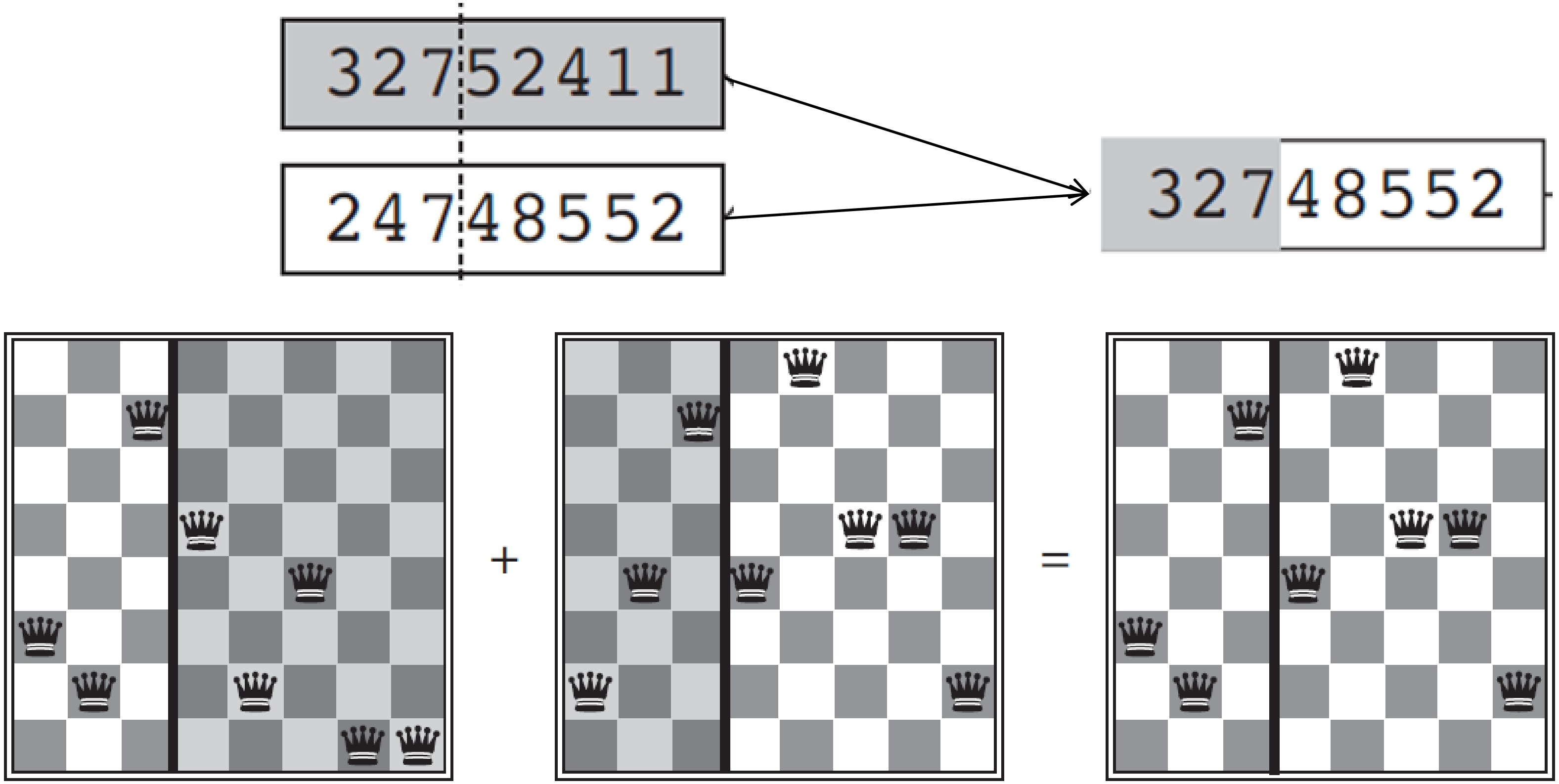


8 Queens solved with Genetic Algorithms

- Chain of numbers giving the position of the queens in the columns
- Fitness = number of non-attacking pairs of queens
  - the higher the value, the better the configuration
  - a solution has value 28



Crossover



Are Genetic Algorithms good for Optimization?

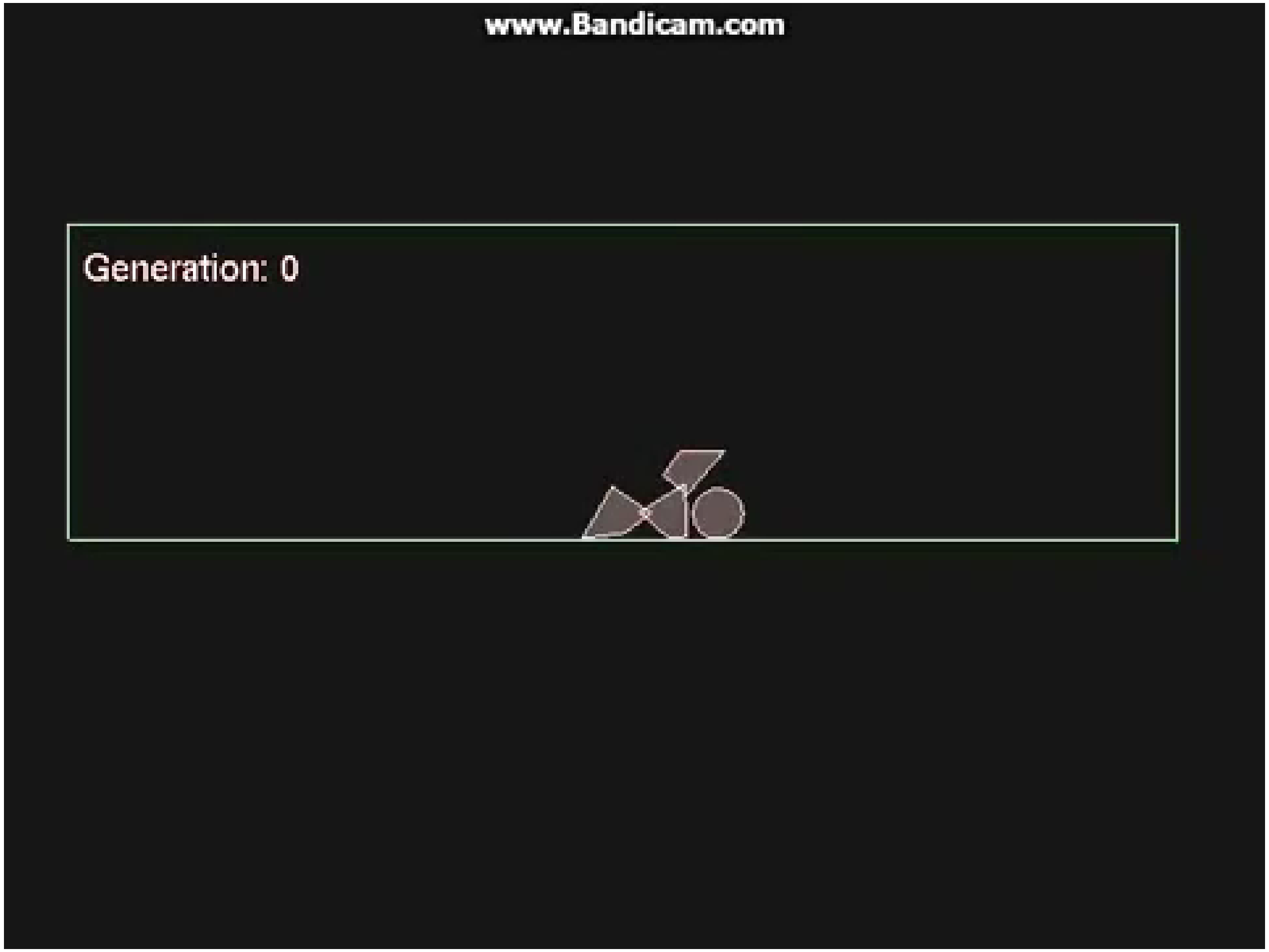
- **NO! -** otherwise, we would all be equal
- GAs are suitable to generate a variety of good solutions, but not in finding the optimal solution
  - evolution ensures the survival of the fittest under changing conditions

A mixability theory of the role of sex in evolution

Adi Livnat[*†] [‡], Christos Papadimitriou [‡], Jonathan Dushoff [§] and Marcus W. Feldman [¶]

evolution of sexual species does not result in maximization of fitness, but in improvement of another important measure which we call mixability: The ability of a genetic variant to function adequately in the presence of a wide variety of genetic partners…

Genetic Algorithm: Learning to Jump over a Ball



https://www.youtube.com/watch?v=Gl3EjiVlz_4

(5) Ant Colony Optimization

- So far, always a single agent searches for a solution …
- Why not use several agents and combine their results?

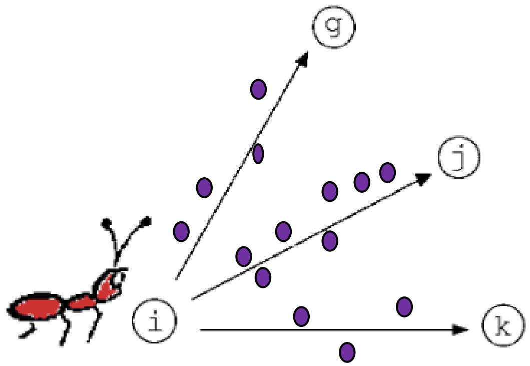
- Form of swarm intelligence
  - ants deposit pheromone on the ground in order to mark some favorable path that should be followed by other members of the colony
- In ACO, a number of artificial ants build solutions to an optimization problem and exchange information on their quality via a communication scheme that is reminiscent of the one adopted by real ants

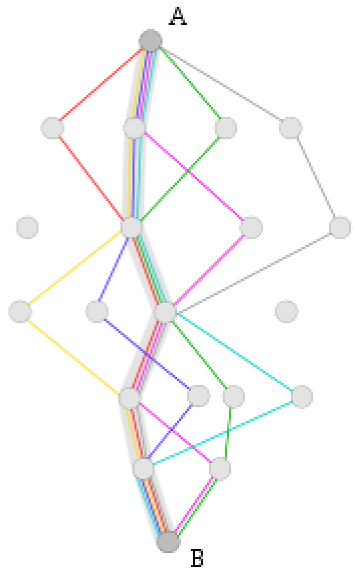
ACO for Traveling Salesperson Problems

At each stage, the ant chooses to move from one city to another according to some rules:

- It must visit each city exactly once
- A distant city has less chance of being chosen (the visibility)
- The more intense the pheromone trail laid out on an edge between two cities, the greater the probability that that edge will be chosen
- Having completed its journey, the ant deposits more pheromones on all edges it traversed, if the journey is short
- After each iteration, trails of pheromones evaporate

➢ Adapts automatically to changing network layouts





An ACO Algorithm

```
procedure ACO_meta-heuristic()
  while (termination_criterion_not_satisfied)
    schedule_activities
      ants_generation_and_activity();
      pheromone_evaporation();
      daemon_actions();   {optional}
    end schedule_activities
  end while
end procedure
```

```
procedure ants_generation_and_activity()
  while (available_resources)
    schedule_the_creation_of_a_new_ant();
    new_active_ant();
  end while
end procedure
```

```
procedure new_active_ant()  {ant lifecycle}
  initialize_ant();
  M = update_ant_memory();
  while (current_state ≠ target_state)
    A = read_local_ant-routing_table();
    P = compute_transition_probabilities(A, M, Ω);
    next_state = apply_ant_decision_policy(P, Ω);
    move_to_next_state(next_state);
    if (online_step-by-step_pheromone_update)
      deposit_pheromone_on_the_visited_arc();
      update_ant-routing_table();
    end if
    M = update_internal_state();
  end while
  if (online_delayed_pheromone_update)
    foreach visited_arc ∈ ψ do
      deposit_pheromone_on_the_visited_arc();
      update_ant-routing_table();
    end foreach
  end if
  die();
end procedure
```

Dorigo, Marco, and Gianni Di Caro. "Ant colony optimization: a new meta-heuristic." Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on. Vol. 2. IEEE, 1999.

Summary

- In very large search spaces heuristic search fails, we thus move to local and stochastic search methods and focus on anytime algorithms
- Stochastic and local search can converge to the optimal solution under very large resources (time, memory)
- UCT is the most popular stochastic search algorithm and has very successful applications in game playing
- Hillclimbing is a simple local search algorithm, which is very powerful when combined with random walks/moves and restarts
- A key to success for stochastic search is to find a good balance between exploration and exploitation
- Meta-heuristic methods (such as genetic algorithms) do **not** guarantee optimality
  - as a result are not suitable for optimization
  - still very popular in practice - they are good for finding a variety of well-fitting solutions

Working Questions

1. How do local and systematic search methods differ?
2. What can we say about the theoretical properties of local search methods?
3. What techniques exist to escape from local optima and plateaus?
4. Why does local search often work well in practice?
5. How does hillclimbing work?
6. Can you explain the main phases and computations of N and Q values of UCT?