Reinforcement learning

Episode that was long overdue...

Planning

Learning Vs planning

Learning

- Black box environment
- Explore through trial and error
- Minimize regret

Planning

- Got environment model
- Search for optimal behavior
- Then act optimally

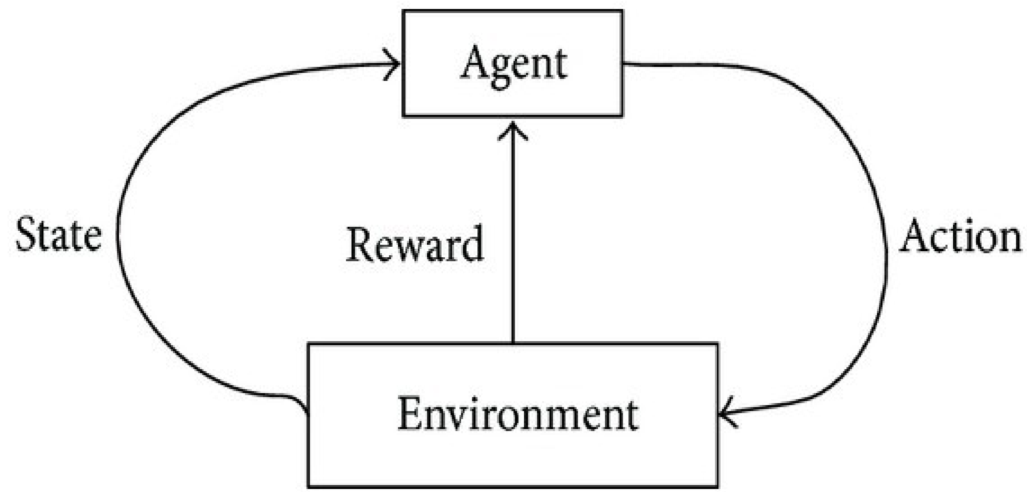
Model-based setup

What we know

- State transitions

$P(s_{next}|s,a)$ or $s_{next}=T(s,a)$

- Rewards $r(s_t,a_t)$



Model-based setup

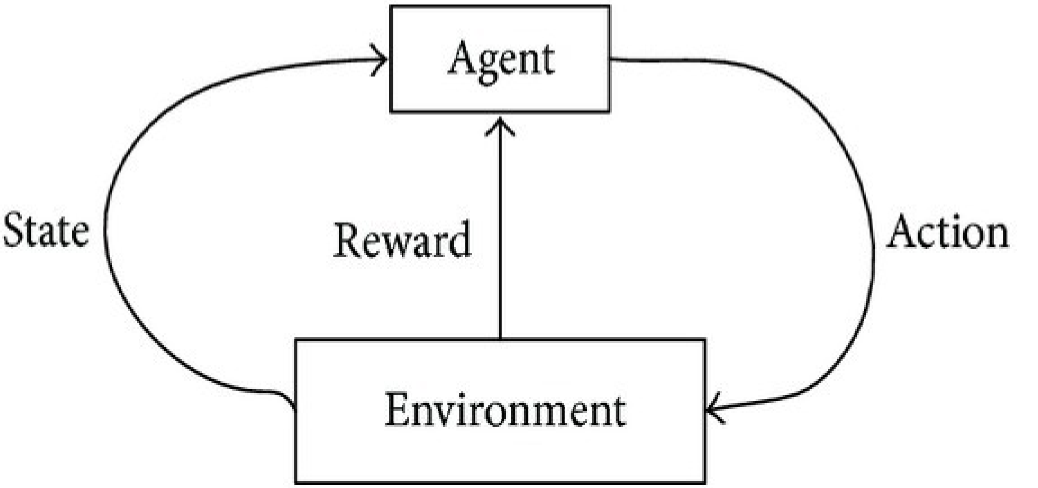
What we know

- State transitions

$P(s_{next}|s,a)$ or $s_{next}=T(s,a)$

Weaker version: we can only sample from P(s'|s,a)

- Rewards $r(s_t,a_t)$



Planning: pathfinding

Further limitations:

- Deterministic $s_{next} = T(s, a)$

- Pay c(s1,s2) for moving s1->s2

- Find shortest route from state A to state B

Trivia: how do we do that?

Planning: pathfinding

Further limitations:

- Deterministic $s_{next} = T(s, a)$
- Pay c(s1,s2) for moving s1->s2
- Find shortest route from state A to state B

Consider c(s1,s2) as a negative reward -r(s1,go_to_s2)

Trivia: how do we do that?

Dynamic programming

Compute following function

$$path(start, end)$$

$$path(a,b) = \min_{v}[path(a,v) + cost(v,b)]$$

Dijkstra's algorithm

- Computes the same function
- Maintains a queue of candidate nodes
- Expands the node with minimal distance to start

Pseudo-code

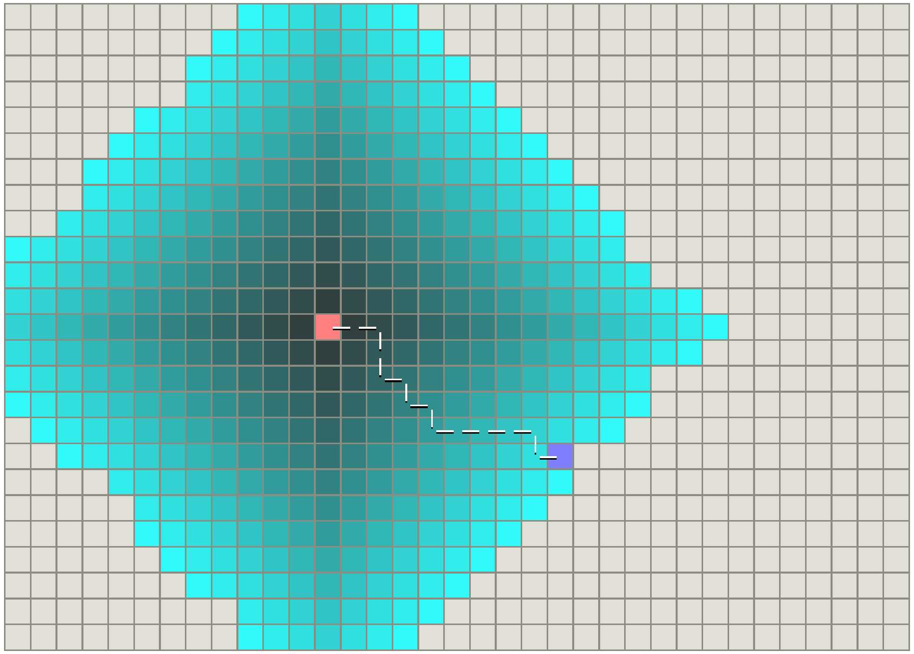
```
distance = {node:inf for each node} #distance to start

distance[start] = 0
fringe = [start]                    #nodes to explore

while True:
  node = fringe.pop_node_with_least_distance()
  if node == end: break

  for neighbor in neighbors(node):
    new_distance = distance[node] + cost(node, neighbor)

    #if we found a better path...
    if new_distance < distance[neighbor]:
      distance[neighbor] = new_distance
      fringe.add(neighbor)
```

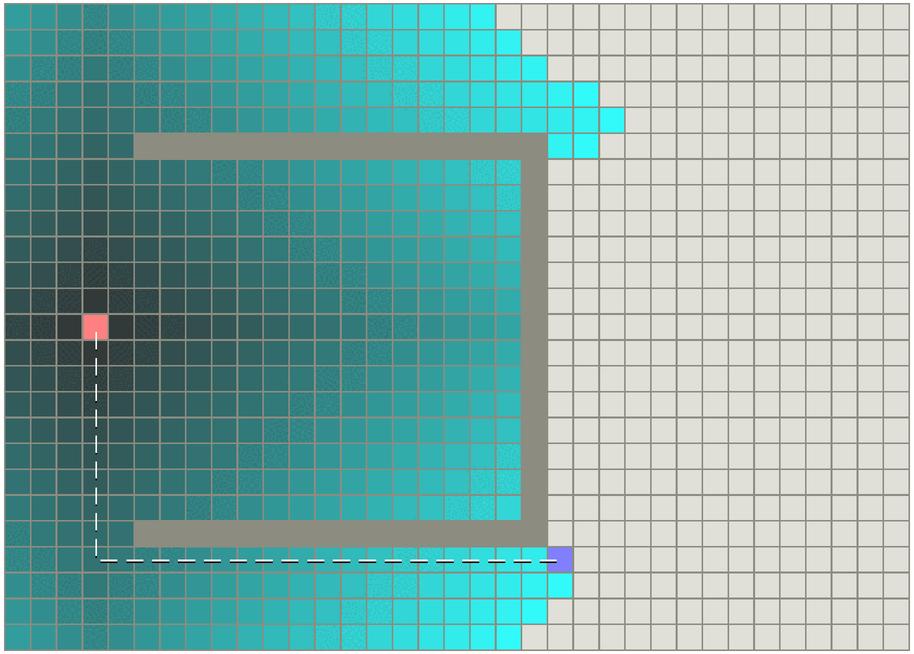
Pseudo-code

```
distance = {node:inf for each node} #distance to start

distance[start] = 0
fringe = [start]                    #nodes to explore

while True:
  node = fringe.pop_node_with_least_distance()
  if node == end: break

  for neighbor in neighbors(node):
    new_distance = distance[node] + cost(node, neighbor)

    #if we found a better path...
    if new_distance < distance[neighbor]:
      distance[neighbor] = new_distance
      fringe.add(neighbor)
```

Trivia: how do we get path(start,end)?

Examples





Blue: viewed nodes, red/violet = start/end, dark-grey = obstacle

A*, informed search

- Heuristic estimate of distance $h(a,b)$

$$h(a,b) \leqslant Path(a,b)$$

- Optimistic path estimate

$$estimate(v) = Path(start, v) + h(v, end)$$

$$\forall v, Path(start, v) + h(v, end) \leqslant Path(start, end)$$

- Pick nodes with least estimate(node)!

A*, informed search

- Heuristic estimate of distance $h(a, b)$

e.g. euclidian distance

$$h(a, b) \leqslant Path(a, b)$$

- Optimistic path estimate

$$estimate(v) = Path(start, v) + h(v, end)$$

$$\forall v, Path(start, v) + h(v, end) \leqslant Path(start, end)$$

- Pick nodes with least estimate(node)!

A*, "informed" search

- Heuristic estimate of distance $h(a,b)$

$$estimate(v) = Path(start, v) + h(v, end)$$

pop_node_with_least_estimate()

```
distance = {node:inf for each node} #distance to start

distance[start] = 0
fringe = [start]                      #nodes to explore

while True:
  node = fringe.~~pop_node_with_least_distance()~~
  if node == end: break

  for neighbor in neighbors(node):
    new_distance = distance[node] + cost(node, neighbor)

    #if we found a better path...
    if new_distance < distance[neighbor]:
      distance[neighbor] = new_distance
      fringe.add(neighbor)
```

A* example

Blue: fringe, red/green= distance, dark-grey = obstacle

Adversarial setup

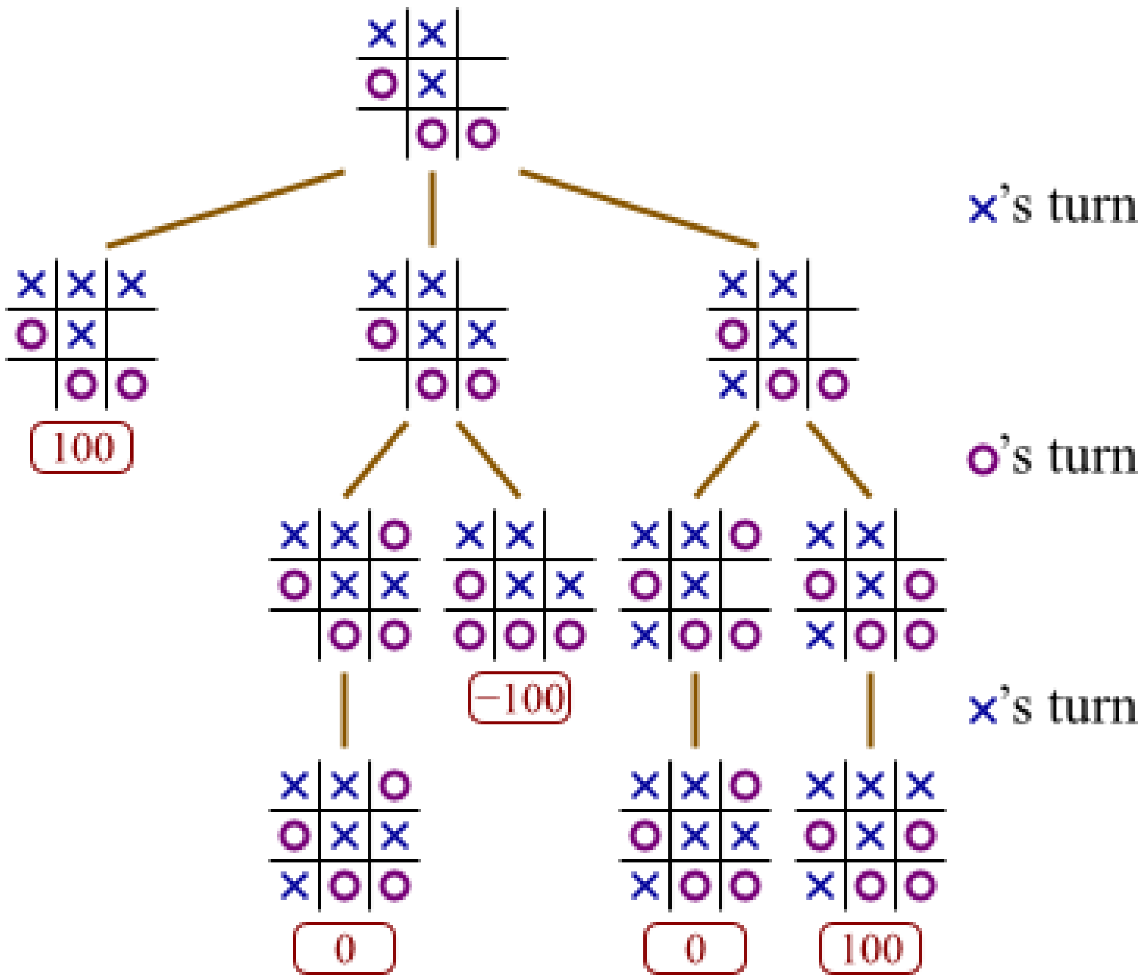
Same as deterministic case, but there's a second agent...

And he's playing against us!

We want highest expected reward.

Examples:
- Any board game: chess, checkers, go
- Pong :)

Adversarial search trees



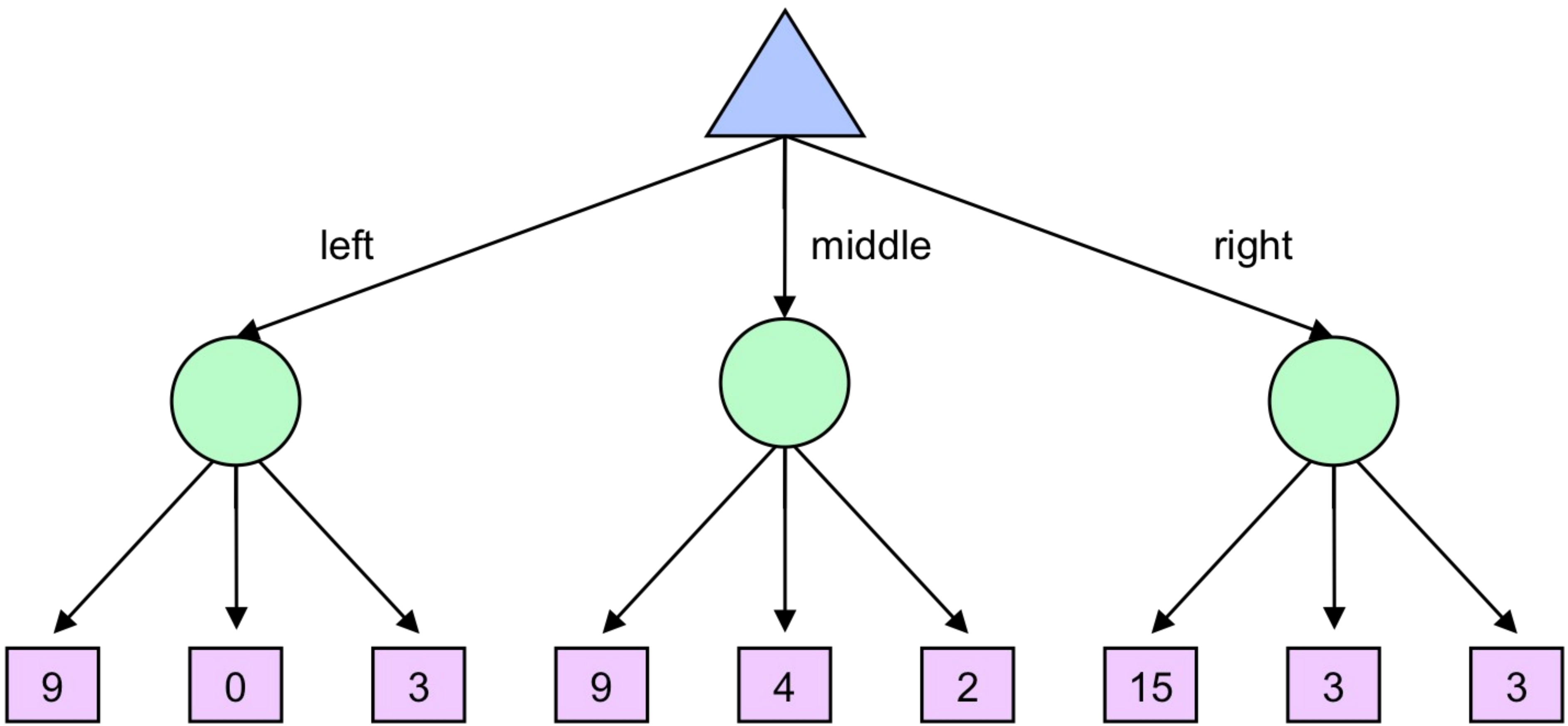
Stochastic setup

Stochastic environment,

$$s \sim P(s_{next} | s, a)$$

We want highest expected reward
or least expected cost

Stochastic search trees



How to evaluate action value?

Large/continuous state space

We can't explore all the nodes.

Need to pick most interesting ones!

Examples:

- ~any practical use case :)
- Atari

Count-based exploration

UCB-1 for bandits

Idea:

Prioritize actions with uncertain outcomes!

Less times visited = more uncertain.

Math: add upper confidence bond to reward.

Count-based exploration

UCB-1 for bandits

Take actions in in proportion to $\tilde{v}_a$

$$\tilde{v}_a = v_a + \sqrt{\frac{2 \log N}{n_a}}$$

Upper conf. bound
for r in [0,1]

- N number of time-steps so far
- n_a times action **a** is taken

Count-based exploration

UCB-1 for bandits

Take actions in in proportion to $\tilde{v}_a$

$$\tilde{v}_a = v_a + \sqrt{\frac{2 \log N}{n_a}}$$

- N number of time-steps so far
- n_a times action **a** is taken

Count-based exploration

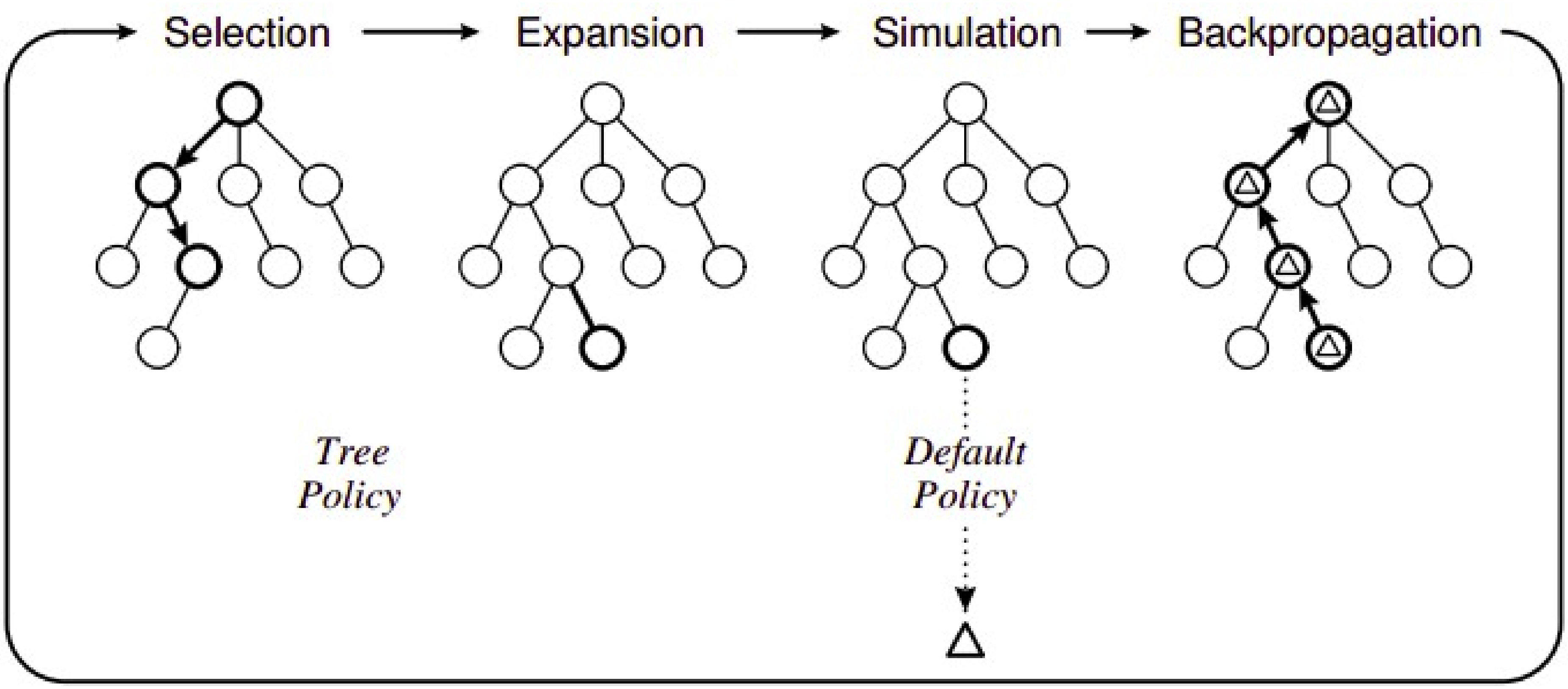
UCB generalized for multiple states

$$\tilde{Q}(s,a) = Q(s,a) + \alpha \cdot \sqrt{\frac{2 \log N_s}{n_{s,a}}}$$

where

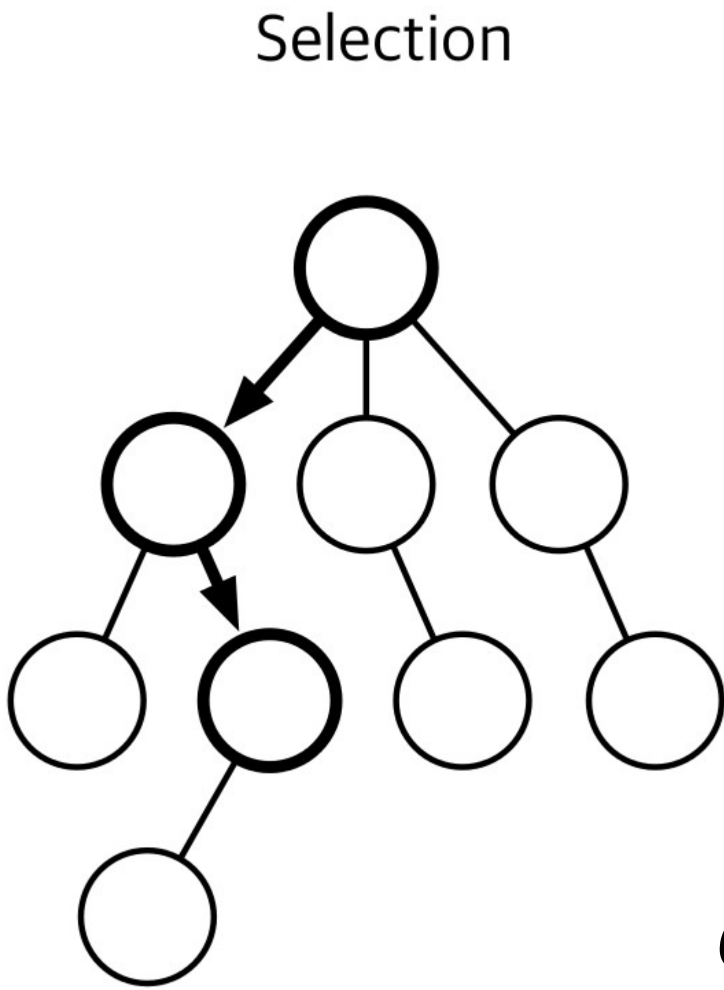
- N_s visits to state **s**
- $n_{s,a}$ times action **a** is taken from state **s**

MCTS



Selection
Expansion
Simulation
Backpropagation
Tree
Policy
Default
Policy

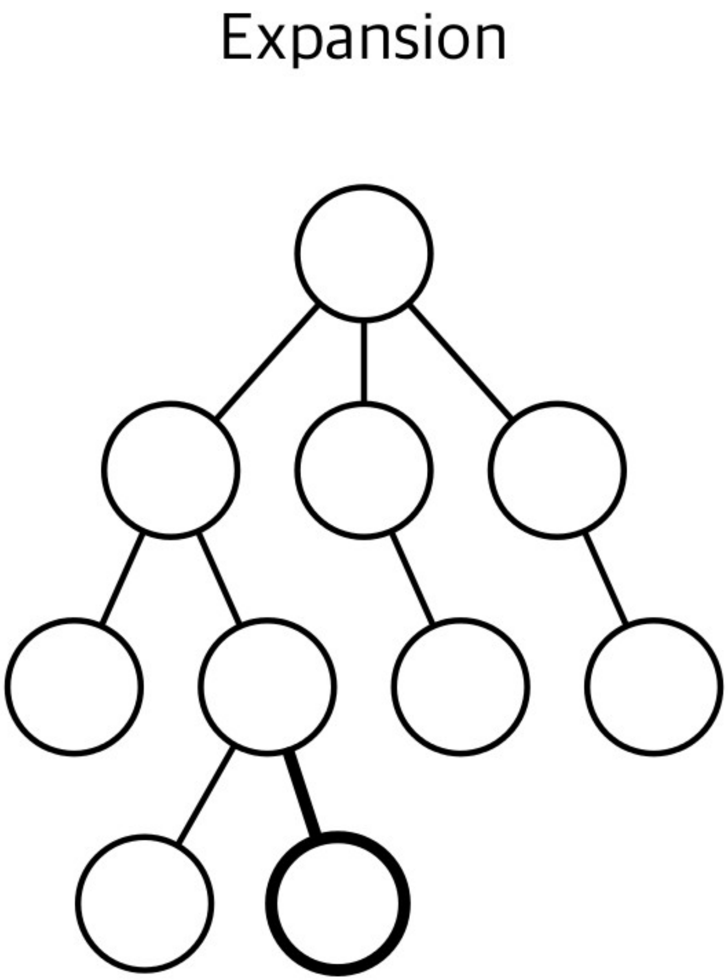
MCTS: selection



Starting from the root, recursively select node with highest ucb-1 score

$$\widetilde{Q}(s,a) = Q(s,a) + \alpha \cdot \sqrt{\frac{2 \log N_s}{n_{s,a}}}$$

MCTS: Expansion

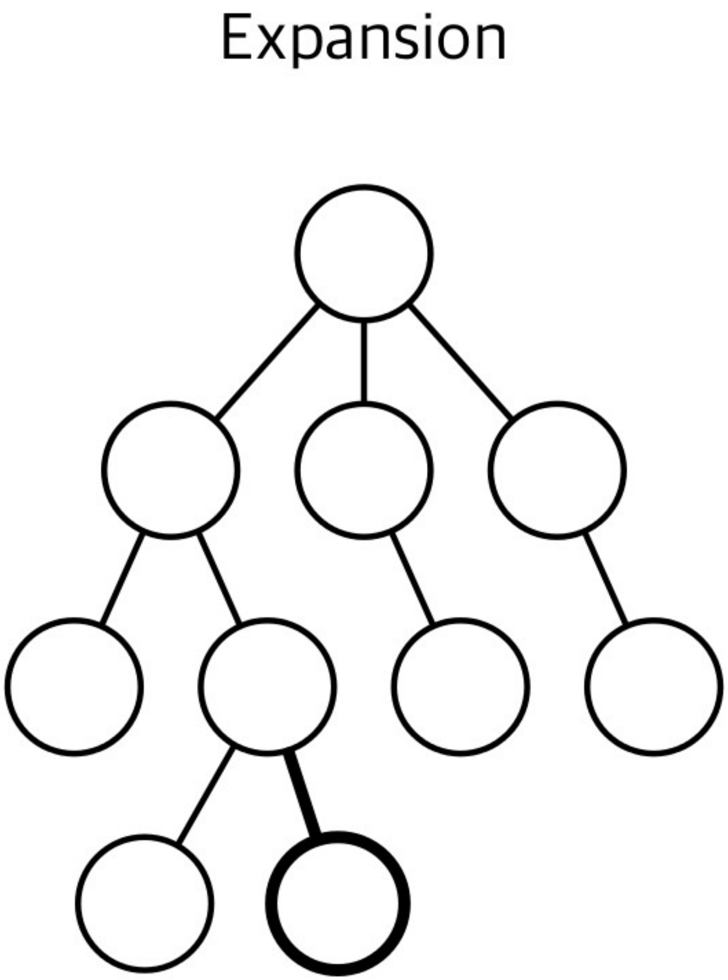


Add one or more children from the chosen node.

Each child is a one-step simulation **s** → **s', a, r**

Simple case: add one node per action.

MCTS: Expansion



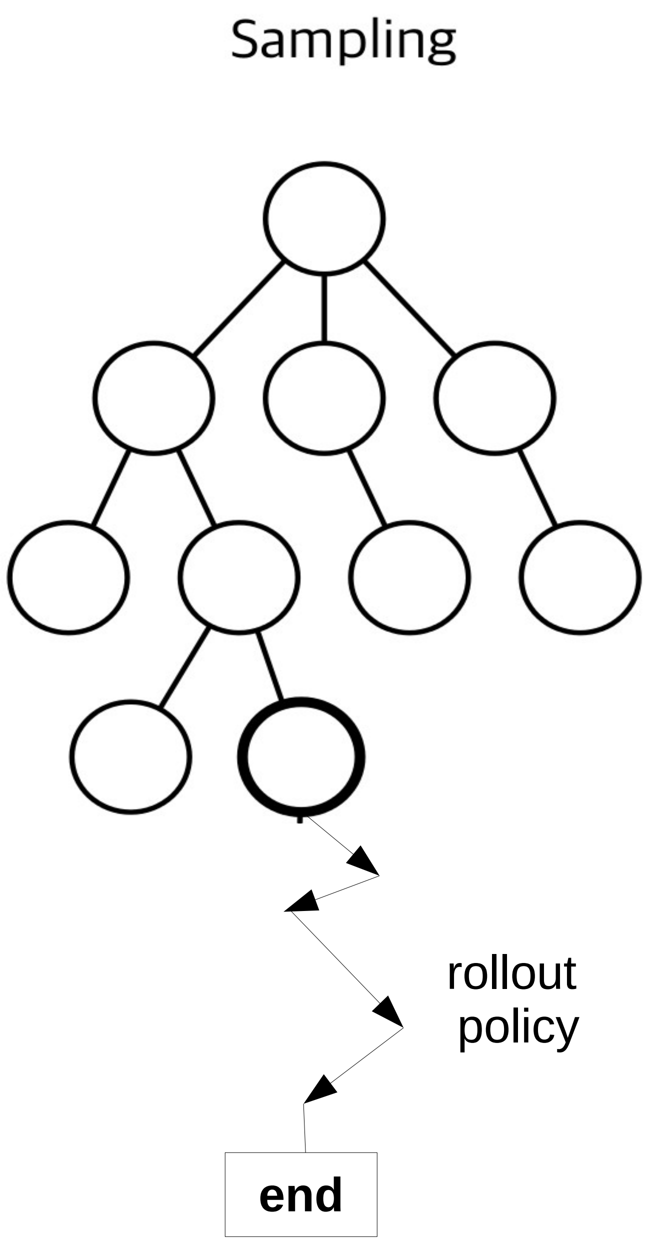
Add one or more children from the chosen node.

Each child is a one-step simulation **s** → **s', a, r**

Simple case: add one node per action.

Any ideas when this is won't work?

MCTS: Rollout (sampling)

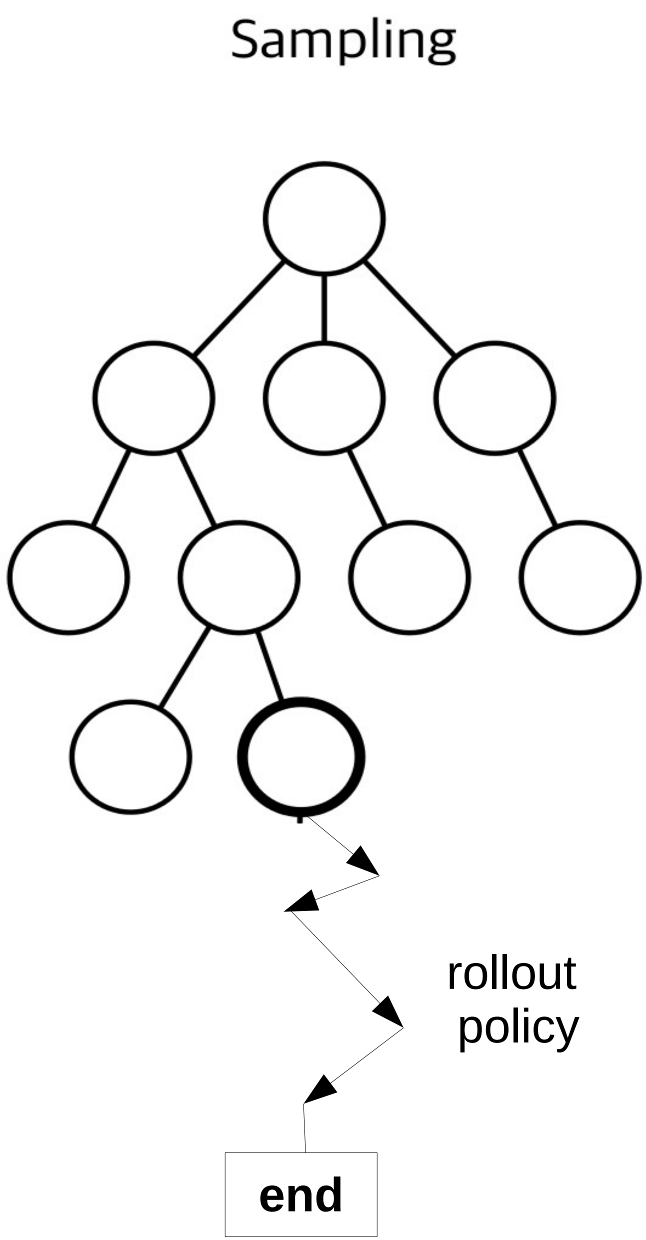


Estimate node value by playing game from that state till the end with simple policy.

e.g. random actions

Remember total reward.

MCTS: Rollout (sampling)



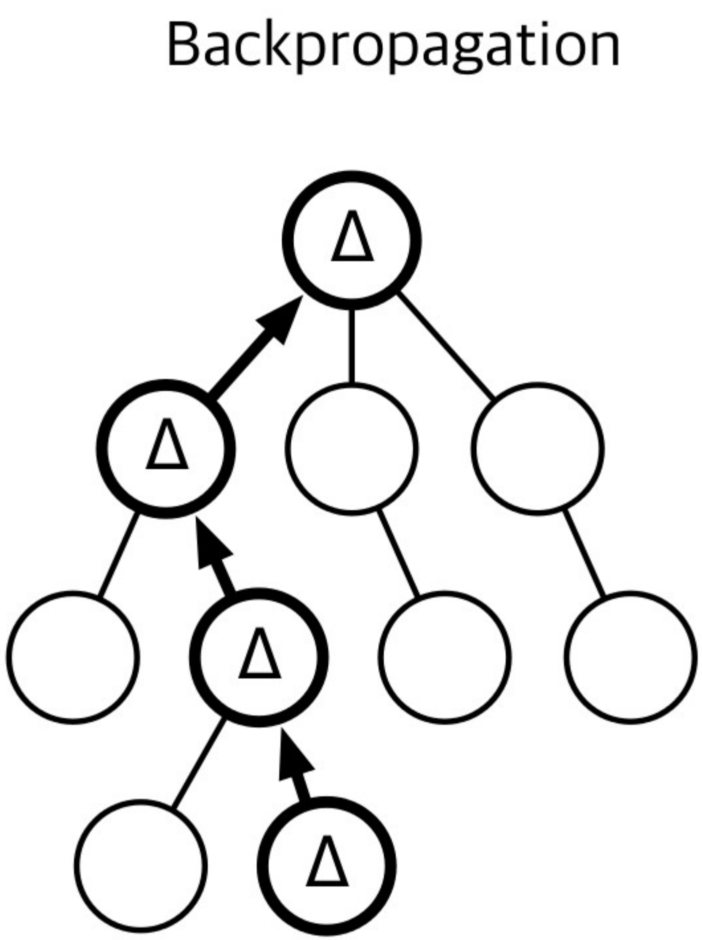
Estimate node value by playing game from that state till the end with simple policy.

e.g. random actions

Remember total reward.

Can we do better than random?

MCTS: Backprop

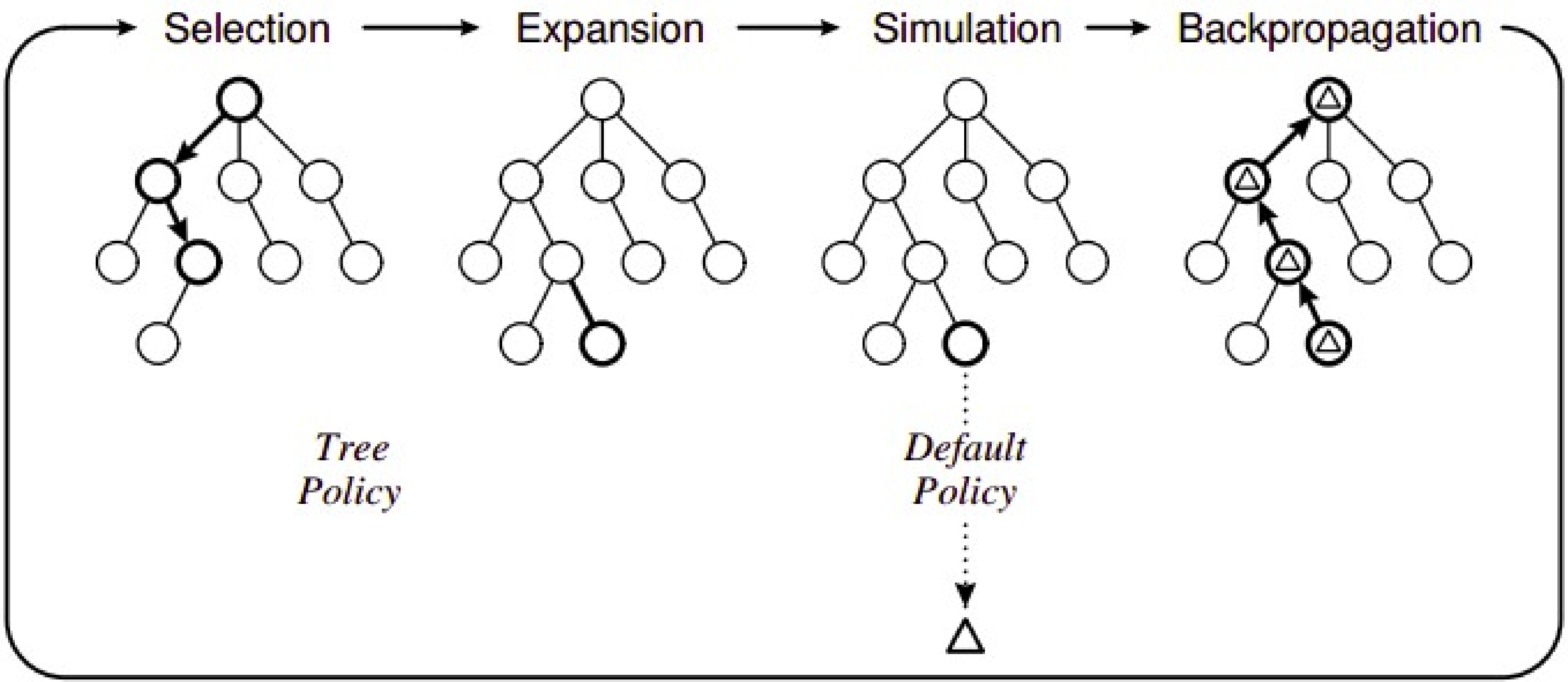


Given rollout reward, update value of leaf and all it's parents.

$$V(parent) = r + \gamma \cdot V(child)$$

Also increment visit counts (N and n_a for ucb-1)

MCTS



How do we pick action from root?

Brace yourselves

And now goes the part with actual cool stuff...