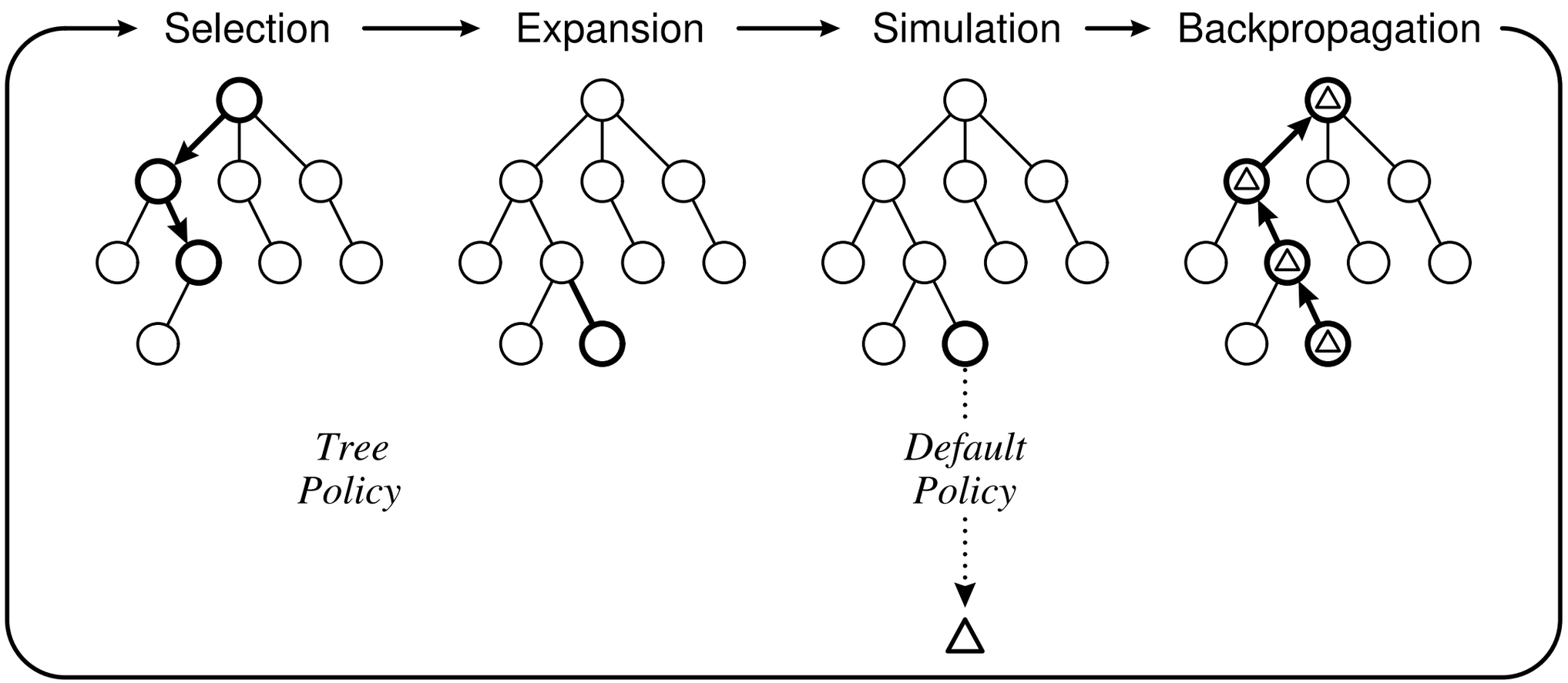
Lecture 14:
MCTS Applications

Artificial Intelligence
CS-GY-6613

Julian Togelius
julian.togelius@nyu.edu

MCTS general idea



- Tree policy: choose which node to expand (not necessarily leaf of tree)

- Default (simulation) policy: random playout until end of game

MCTS success factors

- Reasonably coarse time granularity ("turn-based")
- Any random action sequence leads to the end of the game
  - Sooner rather than later
- Would MCTS work on a game which does not fulfill these constraints?
- If not, could we make it work?

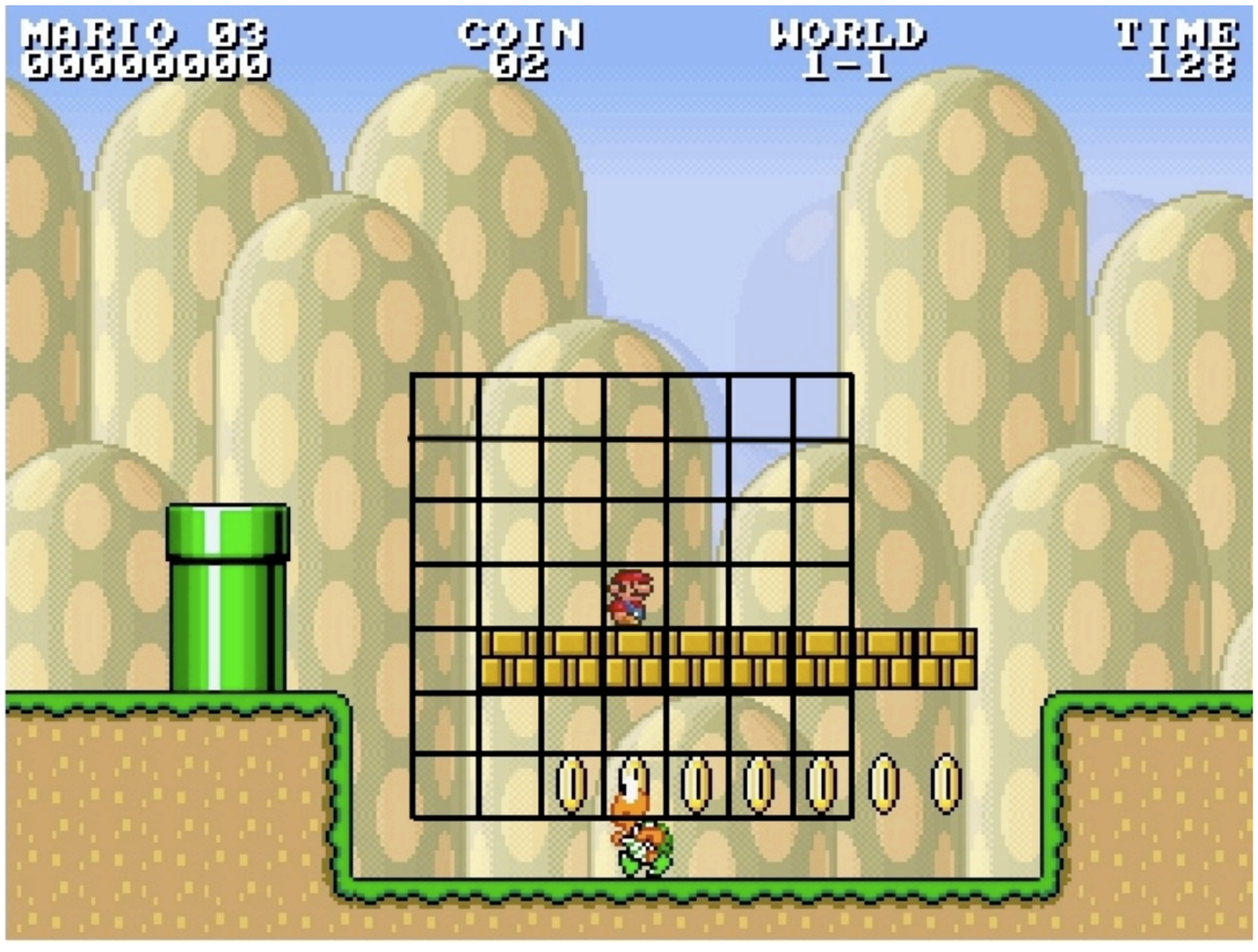
MCTS for real-time games

- Limited roll-out budget
    - Heuristic knowledge becomes important
- Action space may be too fine-grained
    - Take Macro-actions?
- May be no terminal node in sight
    - Again, use heuristic, tune simulation depth
- Next-state function may be expensive
    - Consider making a simpler abstraction

MCTS for Super Mario

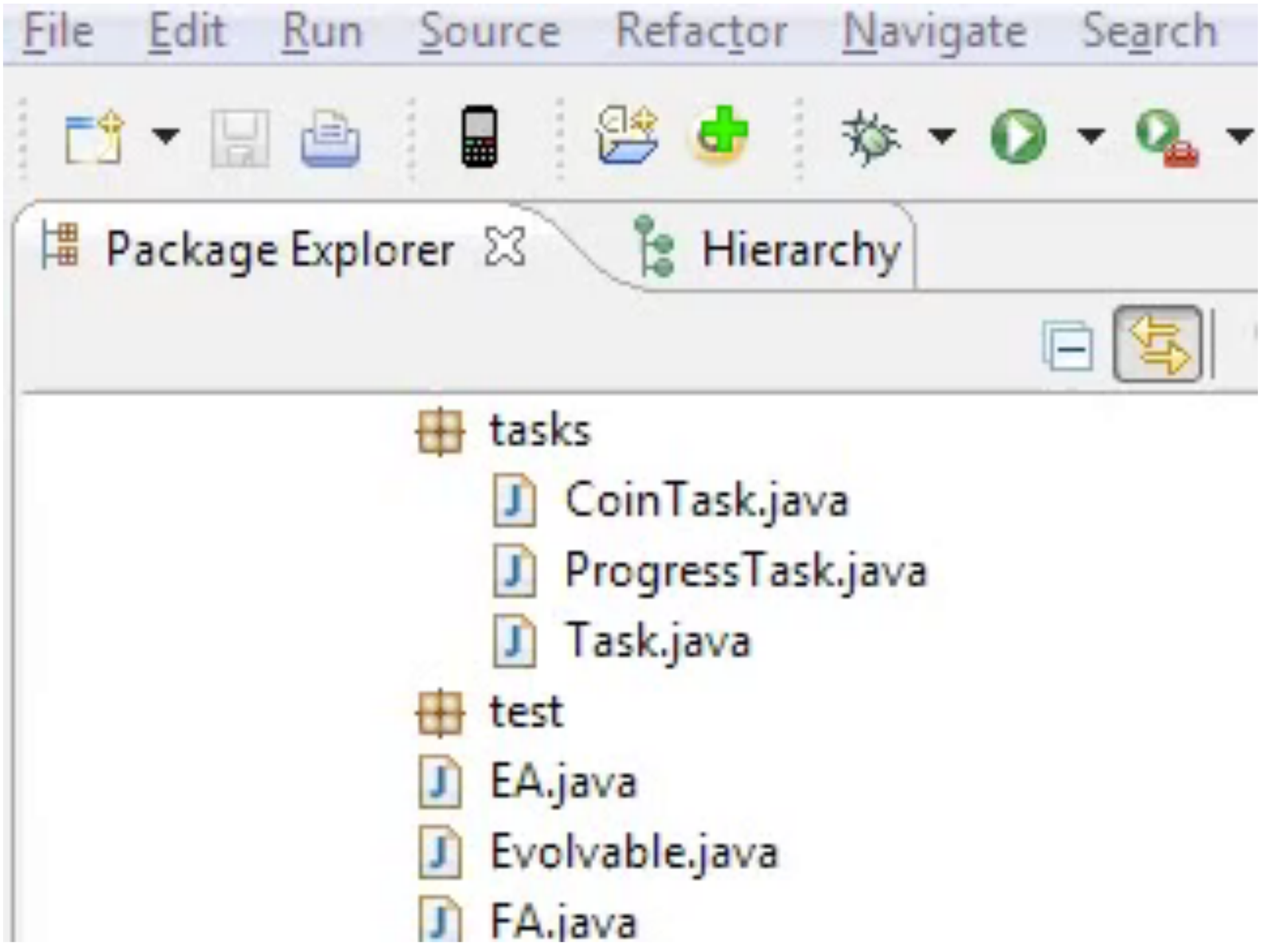
Emil Juul Jacobsen, Rasmus Greve and Julian Togelius

The Mario AI Benchmark



- Reasonably faithful clone of SMB 1/3
- APIs for level generators and AI controllers

A* for Mario


File Edit Run Source Refactor Navigate Search
Package Explorer
Hierarchy
tasks
CoinTask.java
ProgressTask.java
Task.java
test
EA.java
Evolvable.java
FA.java

MCTS for Mario

- We implemented standard MCTS
- Rollouts until the (bitter) end impossible
- Rollout cap (e.g. 6 actions)
- Return progress in x-dimension
  (or -1 if dead)
- Result: Mario is a shortsighted coward

MCTS for Mario

Modifications to MCTS

- Mixmax rewards
- Macro-actions
- Partial expansion
- Roulette wheel selection
- Domain knowledge

Mixmax rewards

- Backing up the average conceals good paths hidden among many bad ones
- Instead, back up a combination of the average and the maximum value achieved by all children of a node

$$exploitation = Q \cdot max + (1 - Q) \cdot \overline{X}_j$$

- Q becomes a “boldness gauge”; 0.25 works.

Mixmax rewards

Macro-actions

- Six or so actions leads to a very short planning horizon
- Instead, make the game take a number of identical actions (e.g. 5) for each node in the search tree
- Leads to a sparser but deeper tree
- Switch to depth 1 when near enemies

Macro-actions

Partial expansion

- As a consequence of the UCB formula, all children of a node are expanded before any grandchildren

$$UCB_j = \overline{X}_j + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{n_j}}$$

- Partial expansion makes it possible to traverse nodes that are partially expanded

$$UCB_c = k + C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{1 + c_n}}$$

Partial expansion

Roulette wheel selection

- The order in which we expand children matters; usually this is done randomly
- Roulette wheel selection: give moves a weight depending on their likelihood to lead to good results
- Choose which move to explore first using a roulette wheel based on these weights

Roulette wheel selection

Domain knowledge

- Remove action combinations that make no sense (e.g. left + right + jump)

- Add hole detection - moves that lead into holes get bad scores early

- Makes modest improvements

Domain knowledge

Individual modifications

Modification	**Mean Score**	**Avg. T Left**
Vanilla MCTS (Avg.)	3918	131
Vanilla MCTS (Max)	2098***	153
Mixmax (0.125)	4093	147
Macro Actions	3869	142
Partial Expansion	3928	134
Roulette Wheel Selection	4032	139
Hole Detection	4196**	134
Limited Actions	4141*	137
(Robin Baumgarten's A*)	4289***	169

Method	Score	T	Method	Score	T
----	3918 †	131	---L	4141* †	137
X---	4093 †	147	X---L	4152* †	147
-M---	3869 †	142	-M--L	4025 †	143
XM---	3922 †	146	XM--L	4043 †	147
-P--	3928 †	146	-P-L	4214* †	146
X-P--	4109* †	140	X-P-L	4278*	150
-MP--	3997 †	134	-MP-L	4156* †	135
XMP--	4166* †	139	XMP-L	4220* †	143
--R-	4032 †	139	--R-L	4132* †	142
X-R-	3786 †	149	X-R-L	4134* †	150
-M-R-	3956 †	145	-M-R-L	4063 †	145
XM-R-	4088 †	146	XM-R-L	4031 †	150
-PR-	4271*	149	-PR-L	4275*	153
X-PR-	4281*	154	X-PR-L	4260*	156
-MPR-	4165* †	135	-MPR-L	3955 †	136
XMPR-	4182* †	141	XMPR-L	4145* †	140
---H-	4196* †	134	---HL	4161* †	139
X--H-	4221* †	145	X--HL	4281*	147
-M-H-	4182* †	142	-M-HL	4251*	141
XM-H-	4197* †	146	XM-HL	4260*	146
-P-H-	2679*†	96	-P-HL	4277* †	138
X-P-H-	3656* †	105	X-P-HL	4277*	147
-MP-H-	2692*†	112	-MP-HL	3970 †	125
XMP-H-	3583* †	105	XMP-HL	4237* †	138
--RH-	4212* †	139	--RHL	4211* †	142
X-RH-	4206* †	148	X-RHL	4195* †	150
-M-RH-	4189* †	143	-M-RHL	4204* †	141
XM-RH-	4240*†	145	XM-RHL	4248*†	147
-PRH-	4268*	148	-PRHL	4284*	152
X-PRH-	4274*	153	X-PRHL	4272*	155
-MPRH-	4071 †	126	-MPRHL	3692 †	126
XMPRH-	4189* †	132	XMPRHL	4061 †	131

Combining the modifications

Everything except Macro

But A* still rules?

- Several MCTS configurations get the same score as A*
- It seems that A* is playing essentially optimally
- But what if we modify the problem?

Making a mess of Mario

- Introduce a action noise: 20% of actions are replaced with a random action

- Destroys A*

- MCTS handles this much better

AI	**Mean Score**
MCTS (X-PRHL)	1770
A* agent	1342**

Summary

- MCTS is a world-class algorithm for e.g. Go
- But makes Mario a short-sighted coward
- We introduced several novel modifications to MCTS with general applicability
- A combination of most of these modifications yielded optimal (?) gameplay
- …and beat A* roundly on noisy Mario!

Let’s go back to Go

Mastering the game of Go with deep neural networks and tree search

David Silver, et al, Nature 2016

What does it do?

- Value networks that evaluate board positions: trained based on supervised learning from human experts
- Policy networks to select moves: trained using reinforcement learning
- Monte Carlo Tree Search

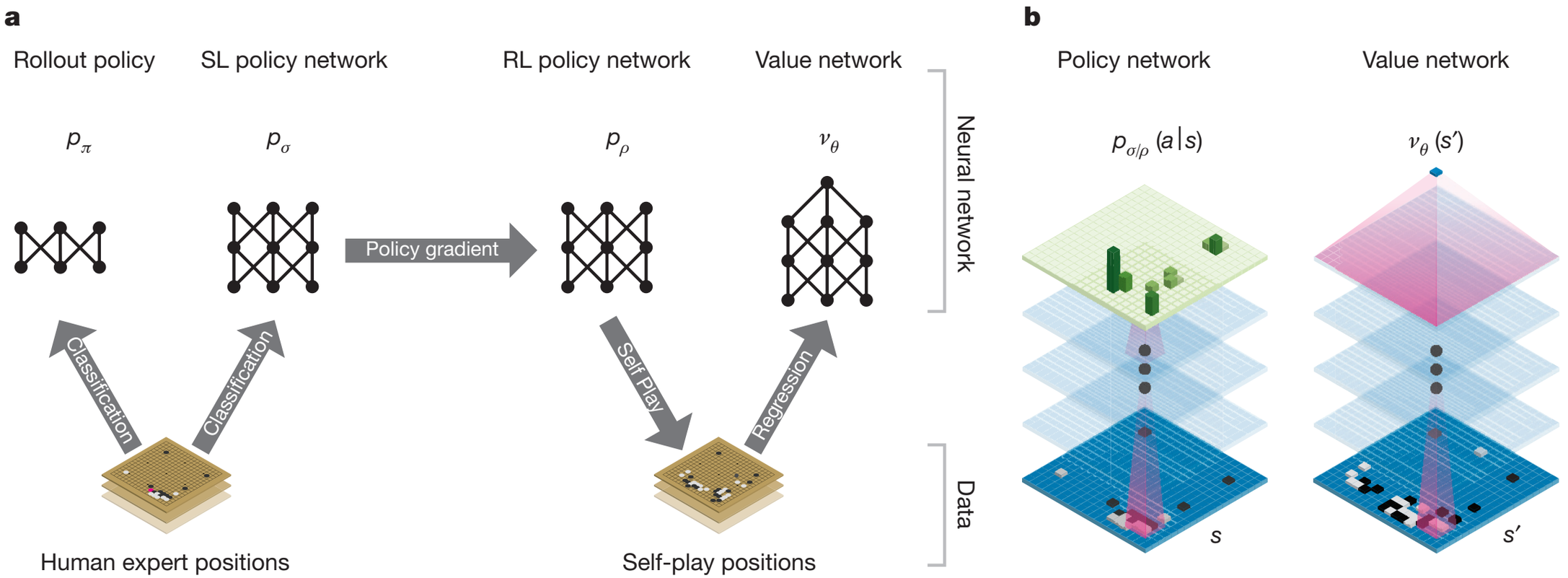


Figure 1 | Neural network training pipeline and architecture. a, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. **b**, Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

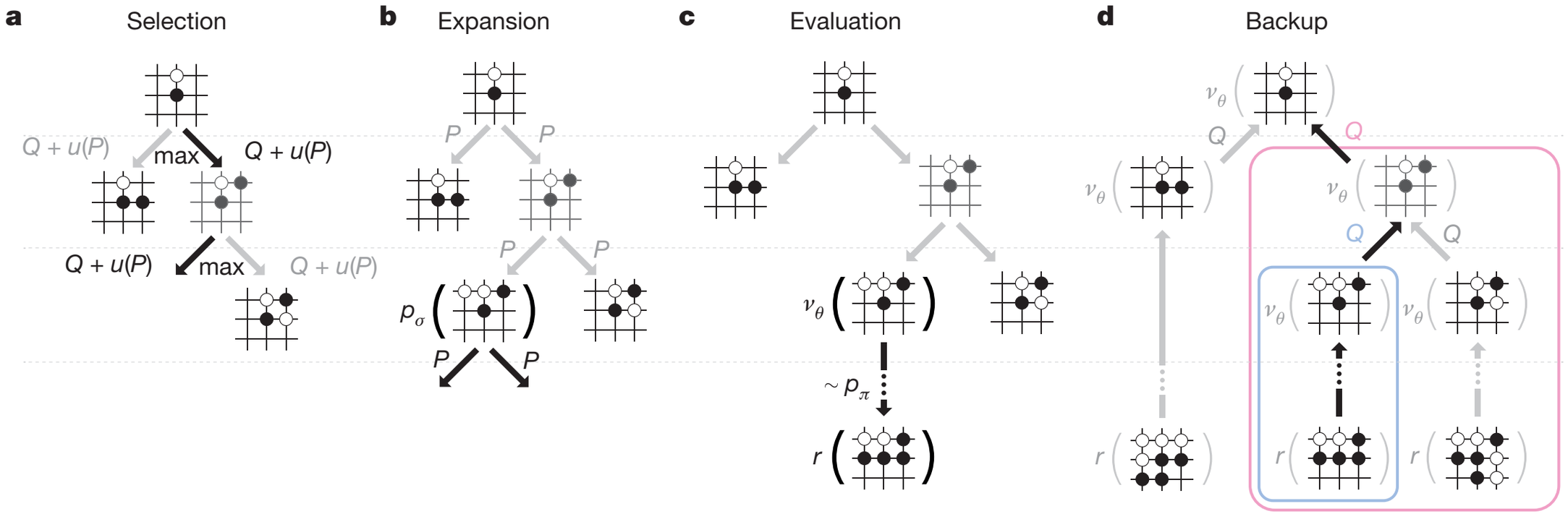
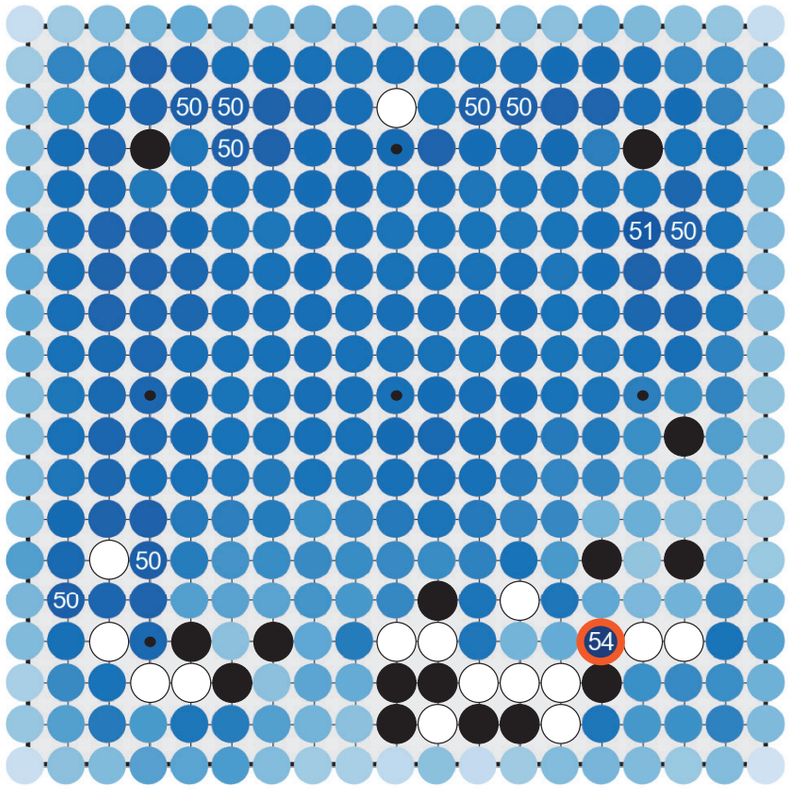
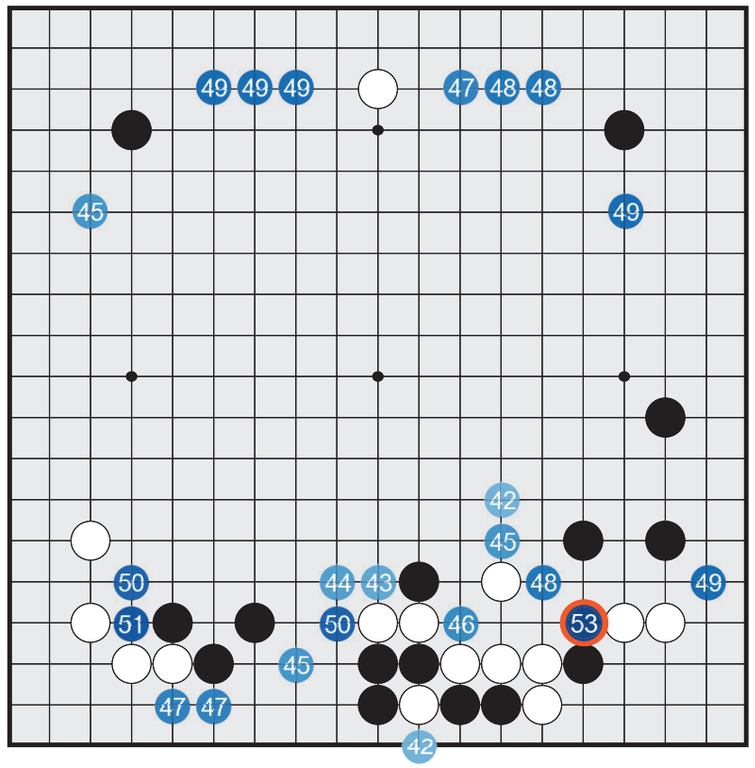


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

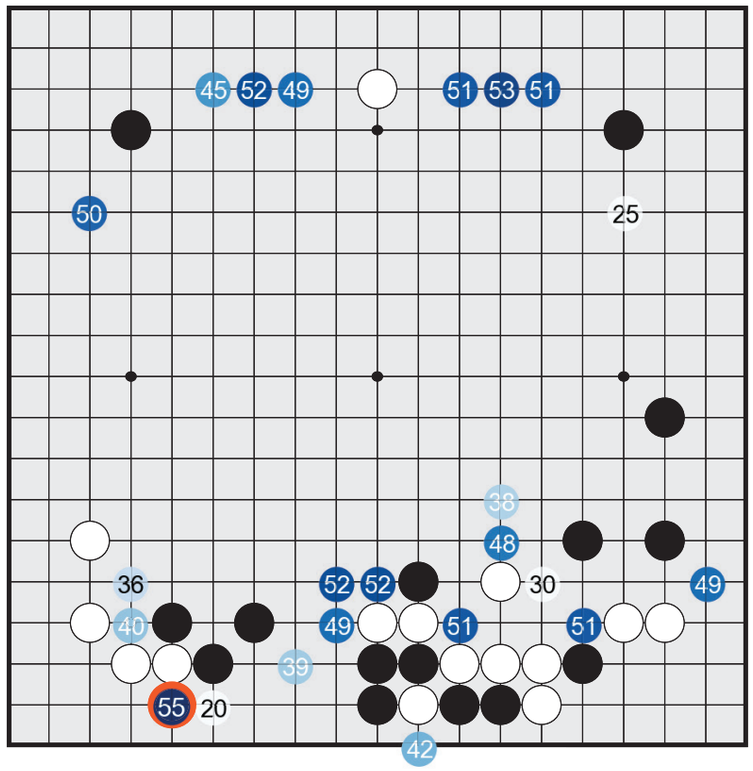
a Value network



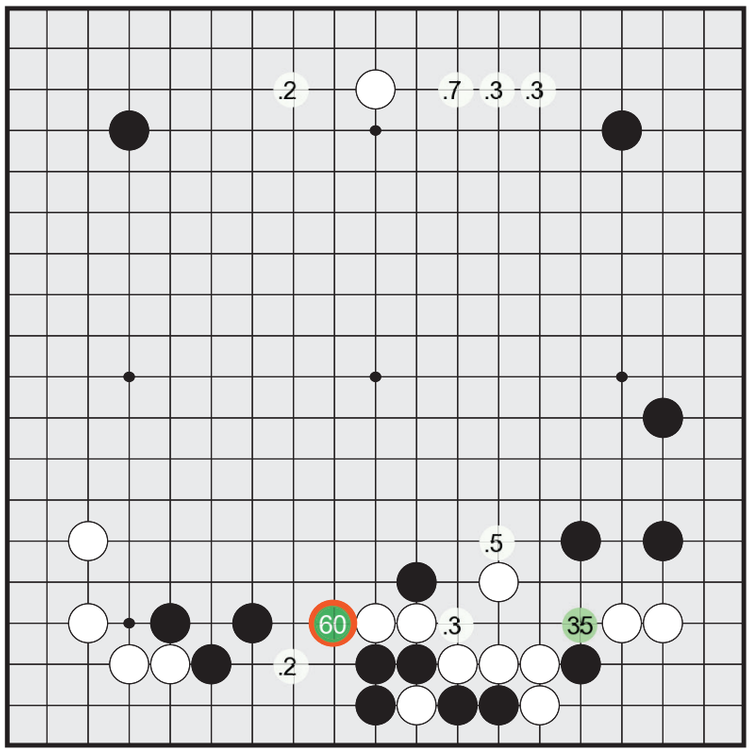
b Tree evaluation from value net



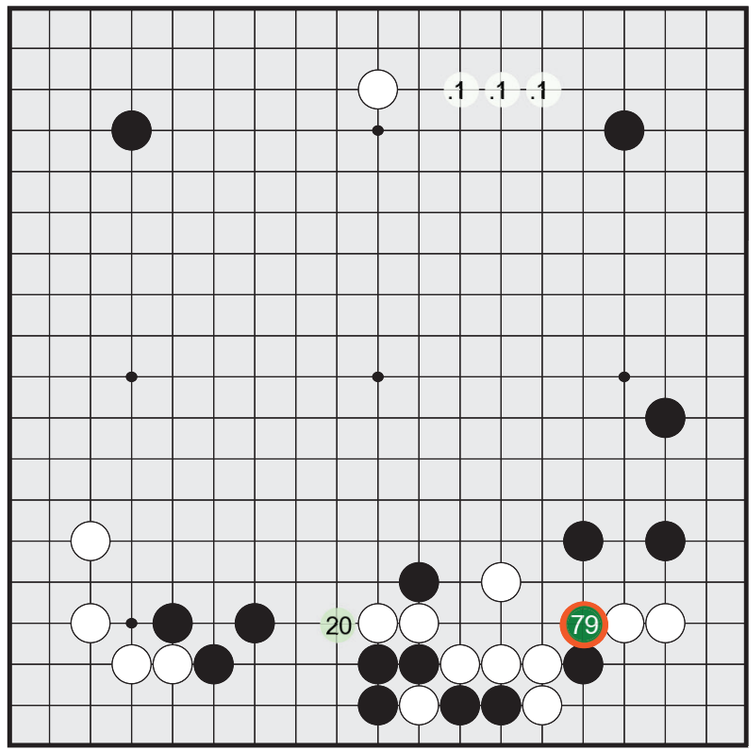
c Tree evaluation from rollouts



d Policy network



e Percentage of simulations



f Principal variation