Monte Carlo Tree Search

Cameron Browne

Computational Creativity Group
Imperial College London
March 2012

Outline

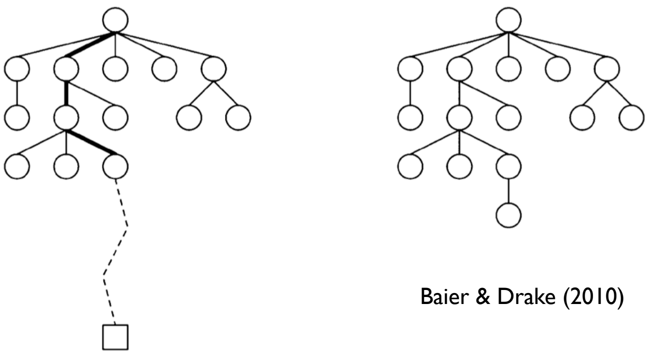
I. Introduction
II. Algorithm
III. Pros and Cons
IV. Variations
V. Enhancements
VI. Demo

I. Introduction

- What is MCTS?
- Game search before MCTS
- The Go revolution (2006)
- Context

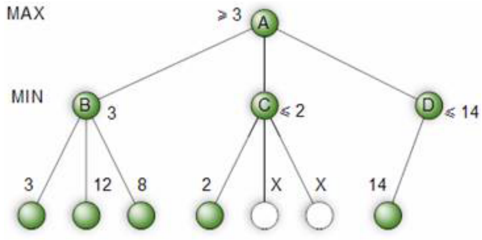
What is MCTS?

- Monte Carlo = random simulation
- MCTS = running random simulations and building a search tree from the results
- Markovian Decision Problem (MDP)
    - Sequences of decisions
    - Any problem phrased as {*state*, *action*} pairs
- Simple!
- But can be powerful



Baier & Drake (2010)

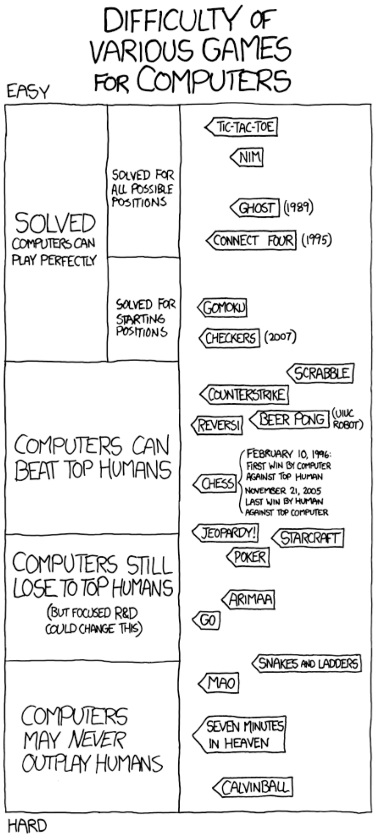
Game Search Before MCTS



Traditional Game Search

- Minimax, alpha beta pruning, etc.
- Works well if:
  - Good heuristic function
  - Modest branching factor
- Chess
  - Deep Blue (Grandmaster level)
  - Shredder (Master level on iPhone)

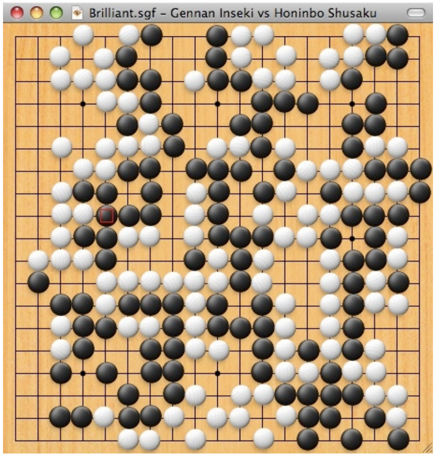
State of the Art



http://www.xkcd.com/1002/

← Traditional search methods insufficient for Go, Arimaa, Poker, StarCraft

The Trouble with Go



- Go is hard!
- 19x19 board
- High move and state complexity:

- Checkers	b.f. ~10	10^{20}
- Chess	b.f. ~40	10^{47}
- Go	b.f. ~275	10^{171}

- No good heuristic function
  - Must play games out
- Studied for decades
  - No strong AI expected for decades to come

The Go Revolution

MoGo (Gelly et al, 2006)

- Challenged human amateurs
- Used MCTS

Computer Go Now

- 9x9 Go: Professional level
- 19x19 Go: Strong amateur level
- Best AIs all use MCTS

MCTS in Other Games

World Champion AIs

- Go (2006-current)
  General Game Playing (2007-current)
  Hex (2008-current)
  etc...

Unofficial World Champions

- Havannah
  Arimaa
  Morpion Solitaire
  etc...
- Chess is the exception

Applications

Computer Go
MoGo
Fuego
CrazyStone
Leela
Many Faces of Go
SteenVreter
Zen

Realtime Games
Ms-PacMan
Real Time Strategy (RTS) Games
Tron
Dead End

Nondeterministic Games
Bridge
Poker
Magic: The Gathering
Backgammon

Solitaire (Puzzle) Games
Sudoku
Kakuro
Crosswords
Morpion Solitaire
SameGame
Bubble Breaker

Connection Games
Hex
Y
Havannah
Renkula
Lines of Action

Combinatorial Games
Amazons
Arimaa
Khet
Shogi
Mancala
Kriegspiel
Clobber
Othello
Blokus
Focus
Connect Four
Sum of Switches

Multiplayer Games
Settlers of Catan

General Game Playing
CadiaPlayer
Ary
Centurio

NON-GAME DOMAINS

Combinatorial Optimisation
Security
Mixed Integer Programming
Travelling Salesman Problem
Physics Simulations
Function Approximation

Constraint Satisfaction

Scheduling
Printer Scheduling
Production Management
Bus Regulation

Sample-Based Planning
Large State Spaces
Feature Selection

Procedural Content Generation
Language
Game Design
Art

Context

- **1928**: Von Neumann proposed minimax tree search
- **1940s**: Monte Carlo methods formalised
- **2006**: Remi Coulom proposed "Monte Carlo tree search"
- What took so long?

Research Interest

- 250+ research papers since 2006
  - Around one per week(!)
- 80+ variations and enhancements already suggested
  - Comparable to entire history of traditional tree search
- Foundations still being laid
  - Many open questions
  - Hot research topic in AI

II. Algorithm

- Operation
- Walkthrough
- Code

Operation

Summary

- Run a number of simulations and build up a search tree according to the results

Assume

- Two-player, zero-sum game (general case)
  1 = win
  0 = draw
  -1 = loss

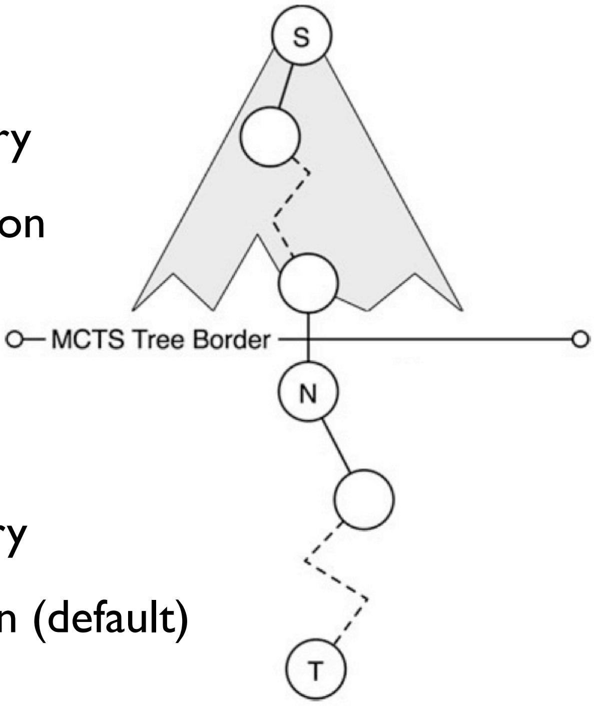
Policies

Tree Policy

- Above the tree boundary
- Intelligent action selection

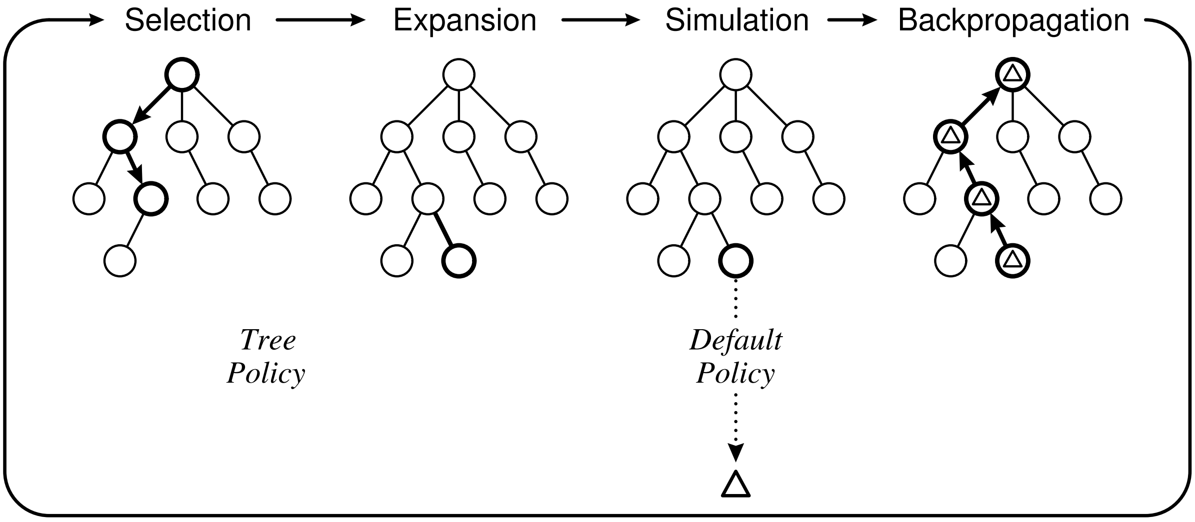
Default policy

- Below the tree boundary
- Random action selection (default)



Finnsson & Bjornsson (2008)

Four Basic Steps



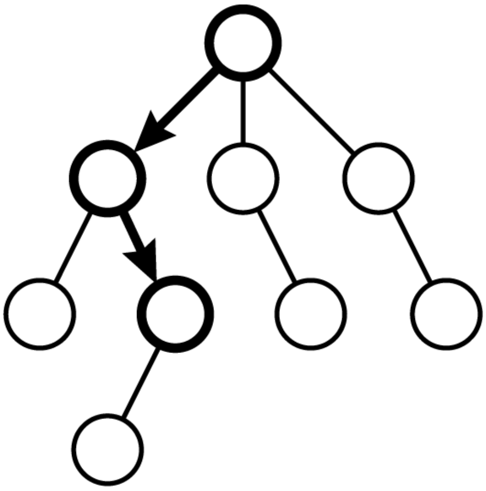
Browne et al (2012)

Each node is a state.

Each iteration adds a node.

Each edge is an action leading to the next state.

1. Selection



Tree Descent

- Start at root (*R*)
- Select most urgent child at each step
- Apply chosen actions
- Stop at tree boundary (terminal state or unexpanded node)

Question: How to select most urgent child?

Upper Confidence Bounds

$$\mathrm{UCB1} = \overline{X}_j + C\sqrt{\frac{2\ln n}{n_j}}$$

- X_j is estimated reward of choice j
- n is number of times parent has been tried
- n_j is number of times choice j has been tried

- Logarithmic *regret* (estimated loss due to suboptimal choices)

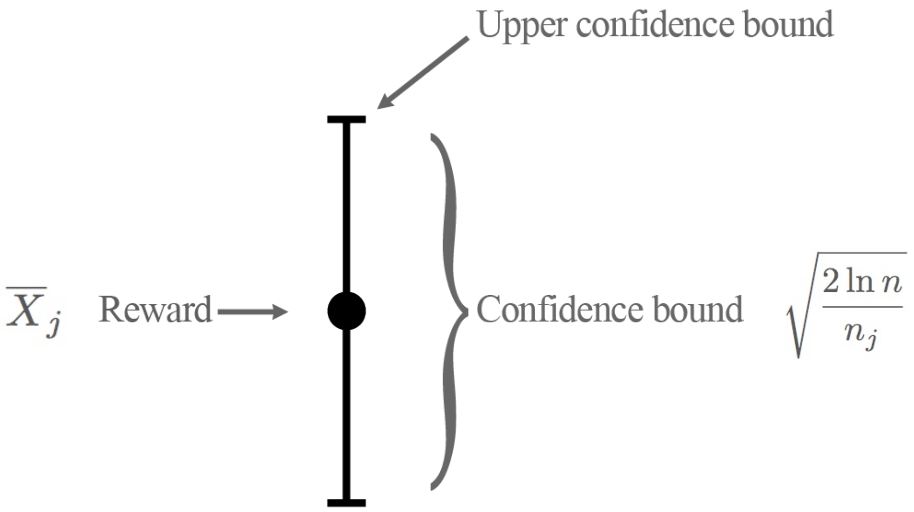
Exploitation vs Exploration

$$\mathrm{UCB1} = \overset{\text{Exploit}}{\overline{X}_j} + C\overset{\text{Explore}}{\sqrt{\frac{2\ln n}{n_j}}}$$

- **Exploitation**
  - Emphasises reward
  - Focusses search
- **Exploration**
  - Encourages exploration of less-tried nodes
  - Reduces effect of unlucky playouts
- Exploration term C balances exploration vs exploitation

Confidence Bounds

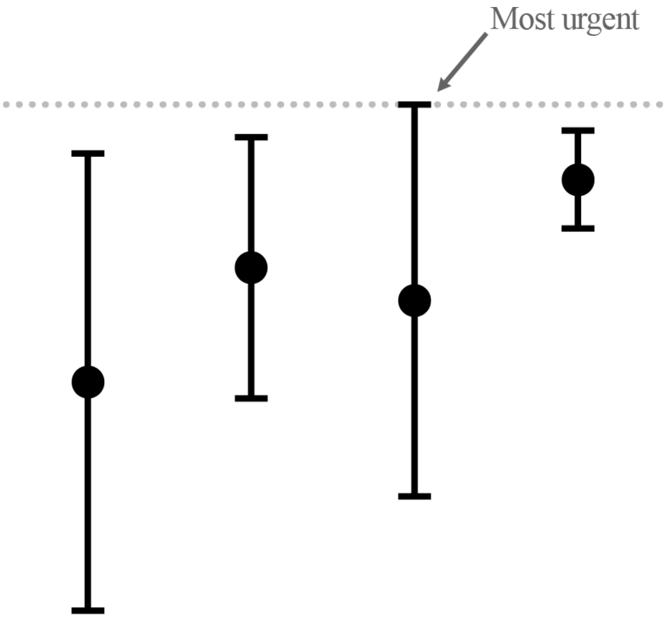
- Confidence in the reward's accuracy

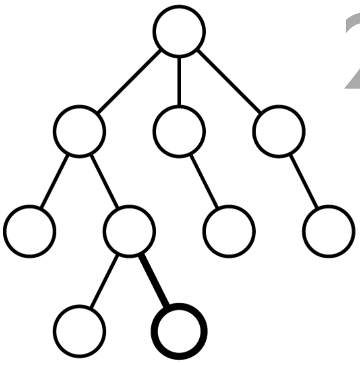


- More visits = tighter bound

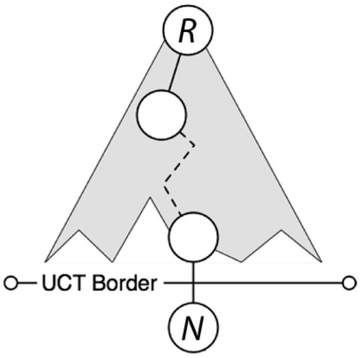
Most Urgent

- Most urgent node has the highest UCB
- **Not** highest reward
- **Not** widest spread



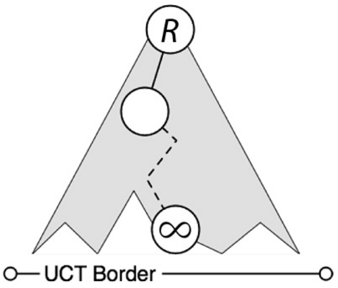


2. Expansion



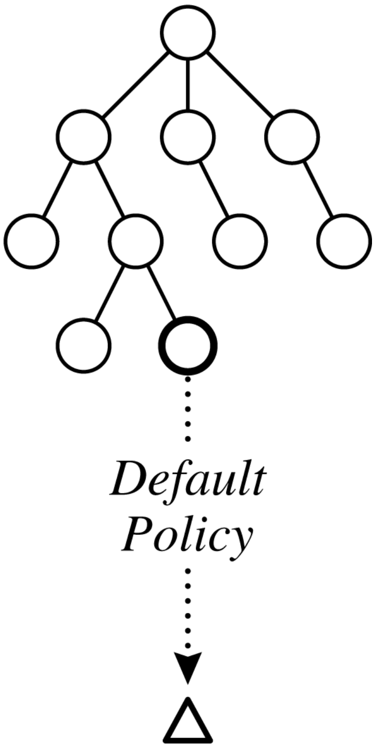
- a) If unexplored child ⇒ expand

  Random order reduces bias



- b) If terminal state ⇒ return

3. Simulation



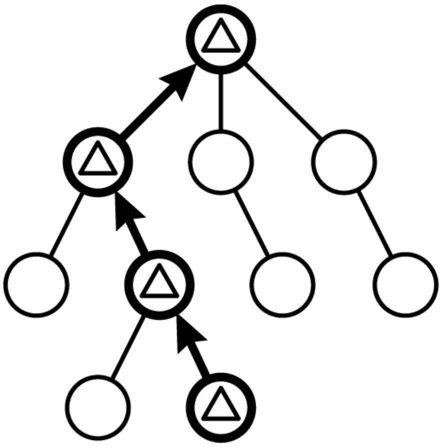
- Play game to conclusion
- Default policy = random playouts

```
while (game not over)
{
  select action a at random
  apply a
}
```

- Return result:
  - Win = 1
  - Draw = 0
  - Loss = -1

---

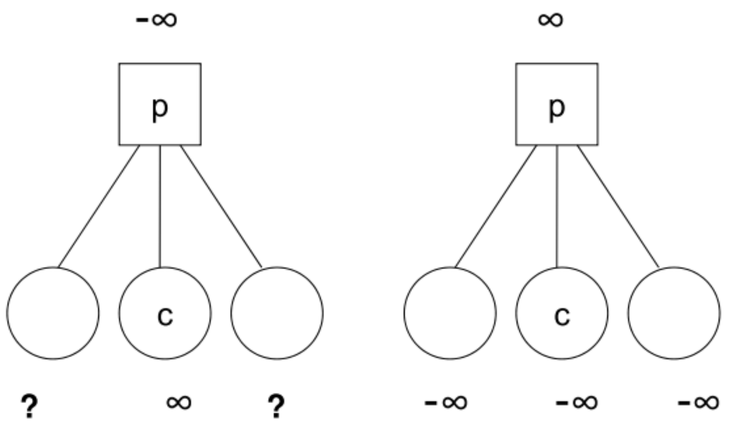
4. Backpropagation



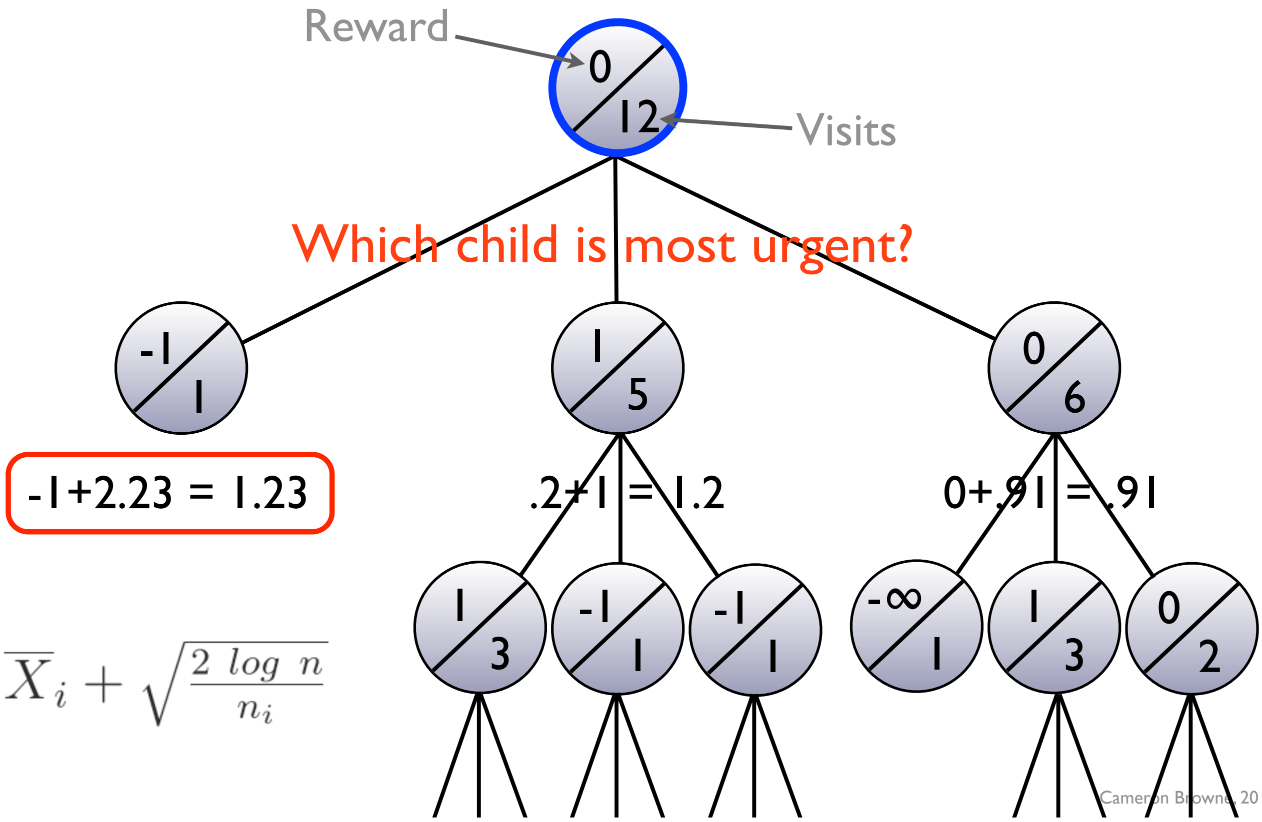
- Update selected nodes with result:
  - Add/subtract result value to node
  - Increment node visit count
- For two-player, zero sum games
  - Win for P_1 is a loss for P_2
  - Negate value with each ply: 1, -1, 1, -1, 1, -1, ...
  - Opponent model

Game-Theoretic Values

- Terminal nodes with known results
  - Handle as per minimax search
- Resolve known values up the tree
  - Convergence to true minimax tree
  - Prune superfluous branches



Example



$$\overline{X}_i + \sqrt{\frac{2\ log\ n}{n_i}}$$

Code

- Java classes
- Pseudocode

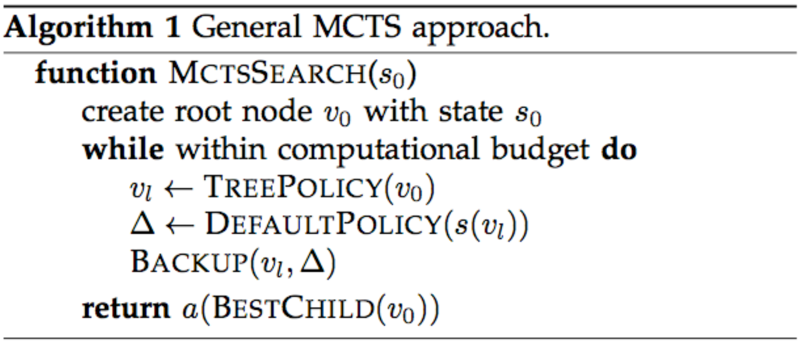
Node Class

```
class Node
{
    int         action;
    int         visits;  // number of times visited
    float       reward;  // accumulated reward value
    Node        parent;  // null if root
    List<Node>  children;

    void update(int value);    // update node and backpropagate to parent
    void addChild(Node child); // add child node
}
```

Game Class

```
class Domain     // typically a game
{
   State state;  // current state

   List<Integer> legalActions();    // list of available actions
   int    applyAction(int action);  // returns winner: None/White/Black/Draw
   int    playout();                // returns: win=+1/draw=0/loss=-1
}
```

MCTS Pseudocode



Algorithm 1 General MCTS approach.

```
function MCTSSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0$))
```

Computational Budget

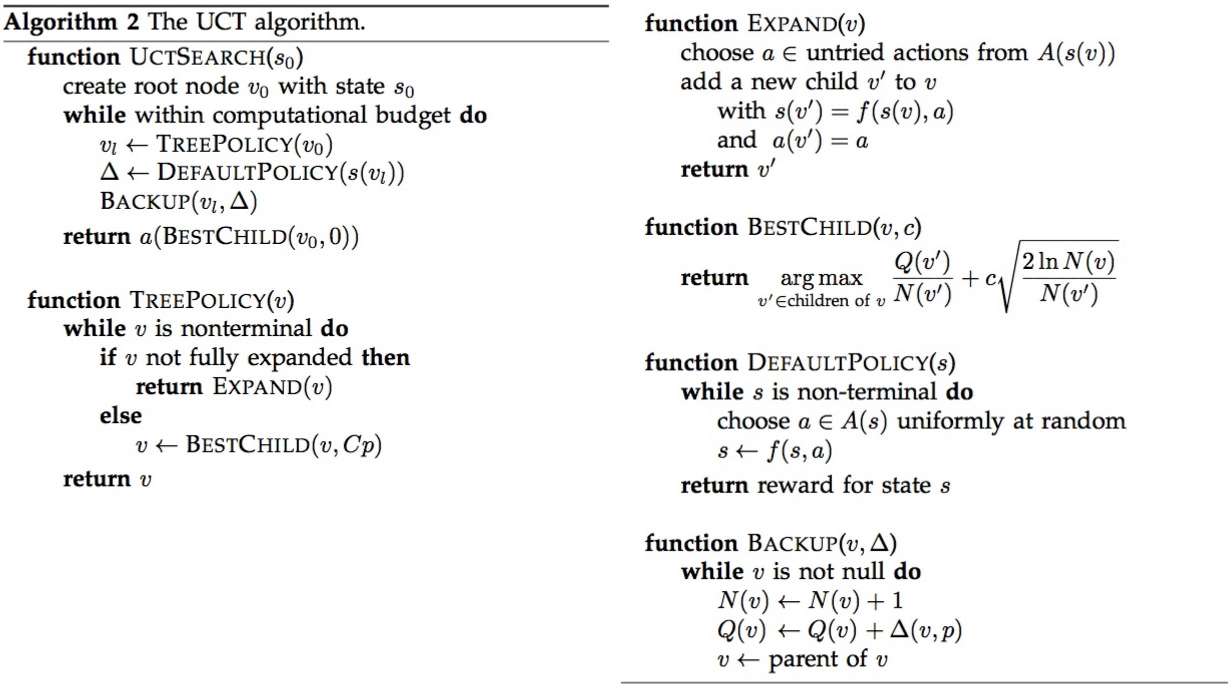
1. *Real time*: for human/tournament play
2. *CPU time*: for experiments
3. *Iterations*: for theoretical comparisons

MCTS and UCT

- MCTS is the general class of algorithms
- UCT is a specific embodiment of MCTS
- UCT = Upper Confidence Bounds for Trees
- UCT = MCTS + UCB

Cameron Browne, 2010

UCT Pseudocode



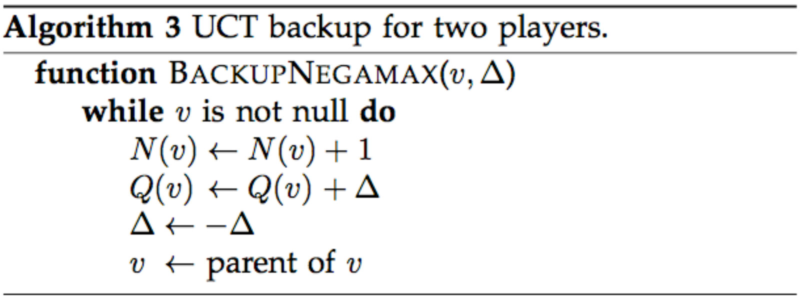
Algorithm 2 The UCT algorithm.

```
function UCTSEARCH($s_0$)
    create root node $v_0$ with state $s_0$
    while within computational budget do
        $v_l \leftarrow$ TREEPOLICY($v_0$)
        $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
        BACKUP($v_l, \Delta$)
    return $a$(BESTCHILD($v_0, 0$))

function TREEPOLICY($v$)
    while $v$ is nonterminal do
        if $v$ not fully expanded then
            return EXPAND($v$)
        else
            $v \leftarrow$ BESTCHILD($v, Cp$)
    return $v$

function EXPAND($v$)
    choose $a \in$ untried actions from $A(s(v))$
    add a new child $v'$ to $v$
        with $s(v') = f(s(v), a)$
        and $a(v') = a$
    return $v'$

function BESTCHILD($v, c$)
    return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

function DEFAULTPOLICY($s$)
    while $s$ is non-terminal do
        choose $a \in A(s)$ uniformly at random
        $s \leftarrow f(s, a)$
    return reward for state $s$

function BACKUP($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta(v, p)$
        $v \leftarrow$ parent of $v$
```

Cameron Browne, 2010

UCT for Two Players

Efficient backup for two-player, zero-sum games



Algorithm 3 UCT backup for two players.

```
function BACKUPNEGAMAX($v, \Delta$)
    while $v$ is not null do
        $N(v) \leftarrow N(v) + 1$
        $Q(v) \leftarrow Q(v) + \Delta$
        $\Delta \leftarrow -\Delta$
        $v \leftarrow$ parent of $v$
```

III. Pros and Cons

Pro

- Aheuristic
- Asymmetric
- Anytime
- Convergence
- Simple

Con

- Weak
- Memory intensive
- Diminishing returns

Aheuristic

No Specific Domain Knowledge

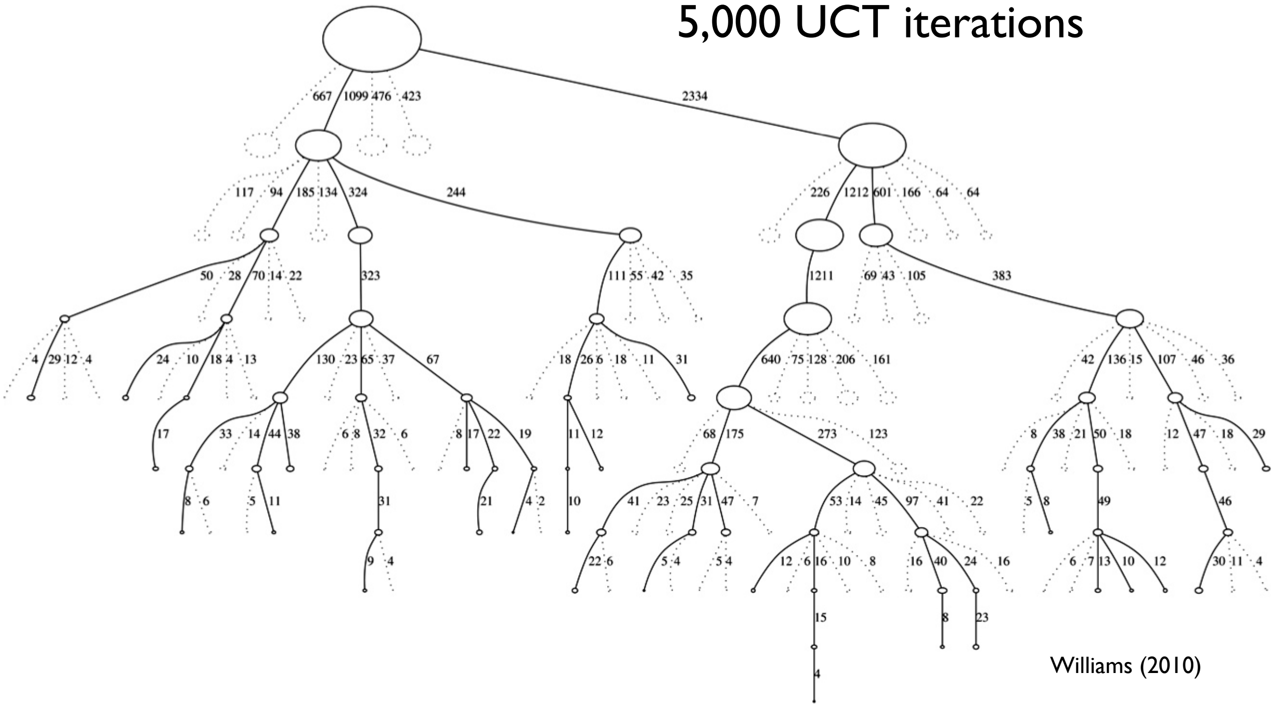
- Available actions for a given state (legal moves)
- Whether a given state is terminal (game over)

No Heuristics

- Intelligent moves with no strategic or tactical knowledge(!)
- Ideal for General Game Players (GGPs)
- Robust to delayed rewards, e.g. Go

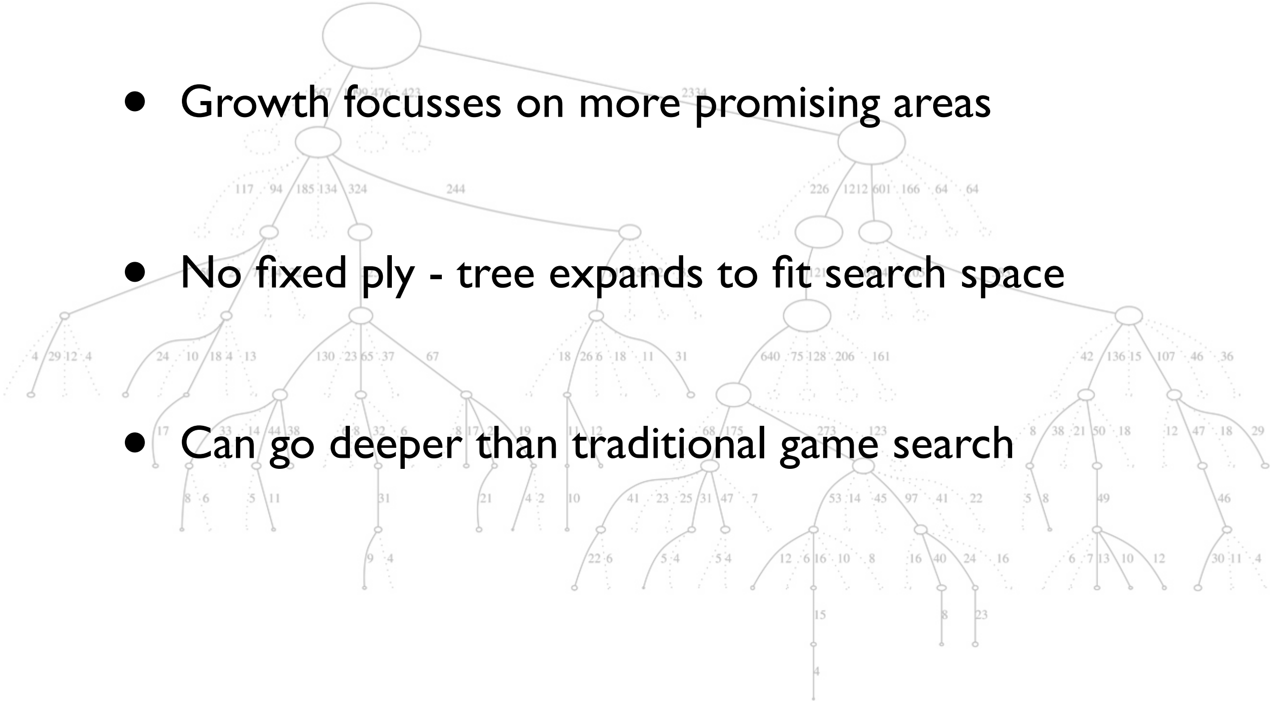
Asymmetric

Kalah (b.f. ~6) Mancala variant
5,000 UCT iterations



Williams (2010)

Asymmetric



- Growth focusses on more promising areas
- No fixed ply - tree expands to fit search space
- Can go deeper than traditional game search

Anytime

- Can stop algorithm anytime to get search result
- Returns immediately
- Continuous board position estimates (not discrete)
  - Better comparison between games

Convergence

- Converges to minimax solution
  - Perfect solution given infinite time
  - Good solution given sufficient time... but when is that?

- Smoothly handles fluctuations in node estimates

Simple

- Easy to code and debug

- Simon Lucas' one-page Java implementation

- www.mcts.ai

Weak

- For simple problems can work extremely well
- For complex problems can be weak unless enhanced
- Generally need to add domain knowledge to work at high level

---

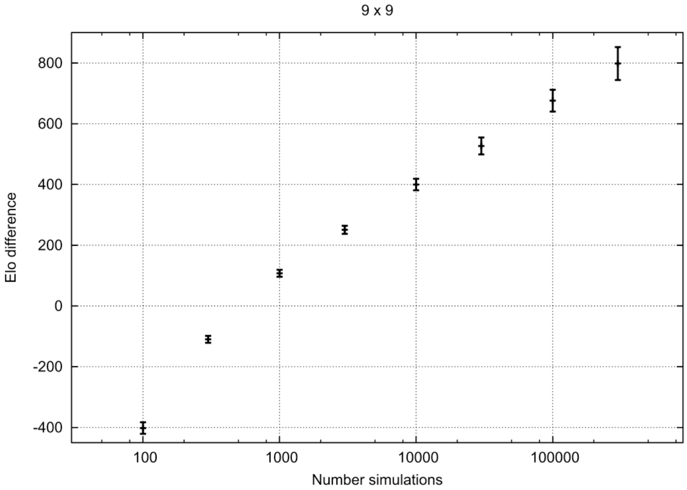
Memory Intensive

- Entire tree must be kept in memory
- But can prune or reuse subtree for subsequent moves

Diminishing Returns

- Twice the playouts ≠ twice the strength!
- 10x playouts → 2x strength

Fuego vs GnuGo



IV. Variations

Flat UCB

UCT

BAST

Learning in MCTS
TDL
TDMC(λ)
BAAL

Single-Player MCTS
FUSE

Multi-player MCTS
Coalition Reduction

Multi-agent MCTS
Ensemble UCT

Real-time MCTS

Nondeterministic MCTS
Determinization
HOP
Sparse UCT
ISUCT
Multiple MCTS
UCT+
MCαβ
MCCFR
Modelling
Simultaneous Moves

Recursive Approaches
Reflexive MC
Nested MC
NRPA
Meta-MCTS
HGSTS

Sample-Based Planners
FSSS
TAG
RRTs
UNLEO
UCTSAT
ρUCT
MRW
MHSP

Useful Variations

UCT

- Upper Confidence Bounds for Trees (UCT)
- Most important variation, has many variations itself
- Used in 90% of MCTS applications

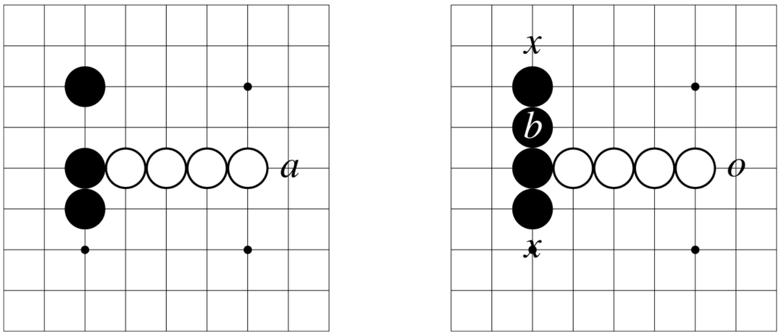
Flat MC and **Flat UCB**

- Random simulation without tree structure
- Good for sanity tests

Opponent Model

Gomoku (5-in-a-row)

- Black to play
- Flat MC prefers losing move *b*
- Why?



Reason

- Flat MC fails to capture opponent model
- UCT correctly chooses *a*

V. Enhancements

- Basic MCTS algorithm is a starting point
  - Usually needs enhancement to perform well

Domain Specific

- Good results
- Only works for current domain

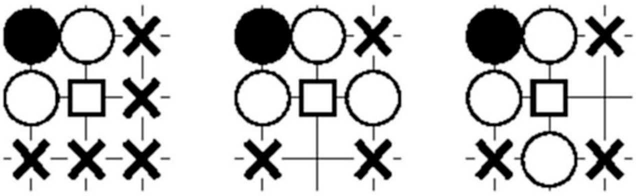
Domain Independent

- Generalise to all problems

Domain Knowledge

Tree Policy

- Prune implausible moves



Default Policy

- "Heavy" playouts
- More realistic results → more reliable node estimates
- Typically known move patterns, e.g. cut moves in Go

Known Enhancements

Bandit-Based
UCB1-Tuned
Bayesian UCT
EXP3
HOOT

Selection
FPU
Decisive Moves
Move Groups
Transpositions
Progressive Bias
Opening Books
MCPG
Search Seeding
Parameter Tuning
History Heuristic
Progressive History

AMAF
Permutation
α-AMAF
Some-First
Cutoff
RAVE
Killer RAVE
RAVE-max
PoolRAVE

Game-Theoretic
MCTS-Solver
MC-PNS
Score Bounded MCTS

Pruning
Absolute
Relative
Domain Knowledge

Simulation
Rule-Based
Contextual
Fill the Board

History Heuristics
Evaluation
Balancing
Last Good Reply
Patterns

Backpropagation
Weighting
Score Bonus
Decay
Transposition Tables

Learning
MAST
PAST
FAST

Parallelisation
Leaf
Root
Tree
UCT-Treesplit
Threading
Synchronisation

Considerations
Consistency
Parameterisation
Comparing
Enhancements

Learning Types

Online Learning

- During play
- e.g. History heuristics, AMAF

Offline Learning

- Before play
- e.g. Opening books, patterns, position values, etc.

History Heuristic

- Keep tally of all moves over all playouts
- Use tally to bias new node values
- Linearly interpolate between historical and MCTS estimates
- Domain independent

AMAF

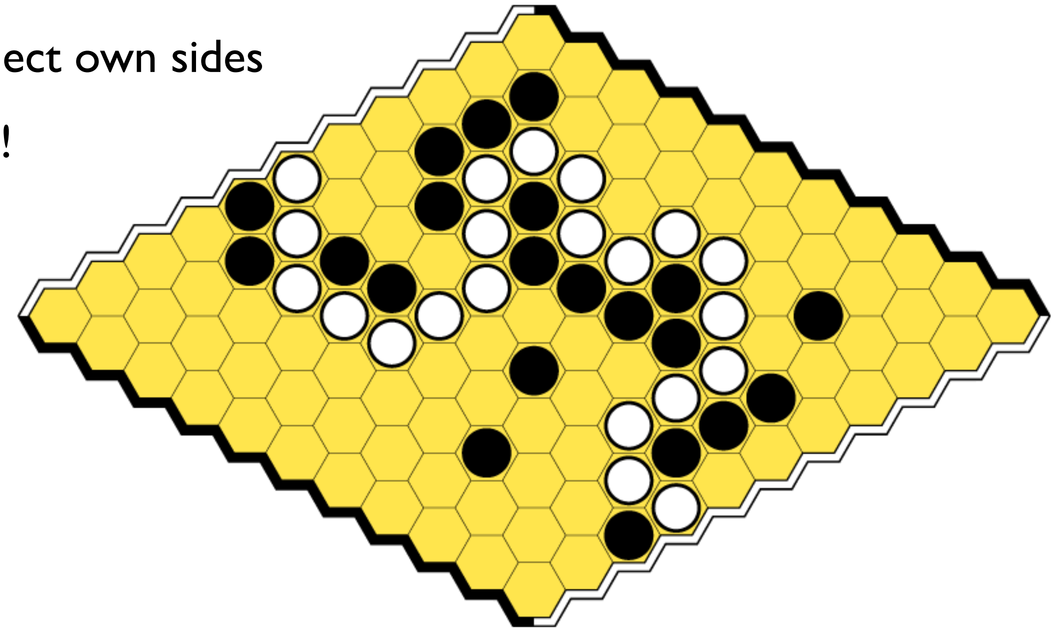
All Moves As First (AMAF)

- Treat each move in playout as next move
- Multiple node updates per iteration
- Rapid Action Value Estimate (RAVE)
- Used in all strong Go players
- May not work for all games, e.g. Othello
- Domain independent

VI. Demo

Hex

- Add a piece per turn
- Connect own sides
- Deep!



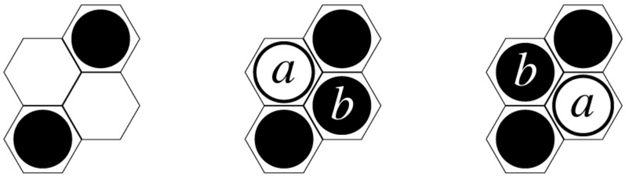
- Exactly one player must win

Domain Knowledge

Complementary Goals

- Random fill → test for win once
- Guaranteed result

Domain Knowledge

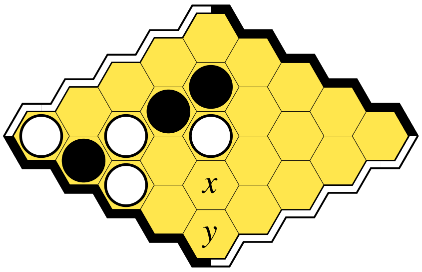


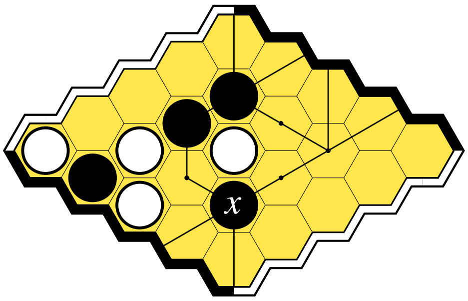
- Bridge pattern
- Heavy playouts → repair bridge intrusions
- Slower but smarter playouts (usual trade-off)

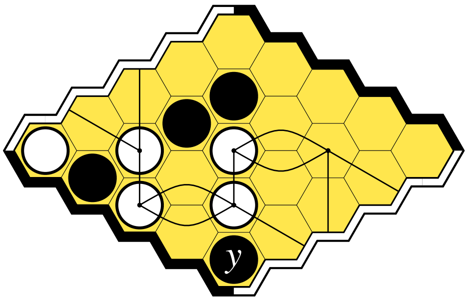
5x5 Puzzle

Problem

- Black to move
- *x* or *y*?







- **Solution:** *x* wins, *y* loses

Demo

Search	**Result**	
Random	5.5%	
Flat MC	0%	
UCT	100%	(~600,000 iters)
$\text{UCT}_{bridges}$	100%	(~10,000 iters)

- Flat MC is worse than random!
- UCT works but UCT with domain knowledge is better

Conclusion

- MCTS has revolutionised computer Go (and other games)
- Application to many domains (not just games)
- Basic algorithm can be weak, but many enhancements
- Hot research topic in AI