CMSC 471
Spring 2024
Midterm Review

KMA Solaiman

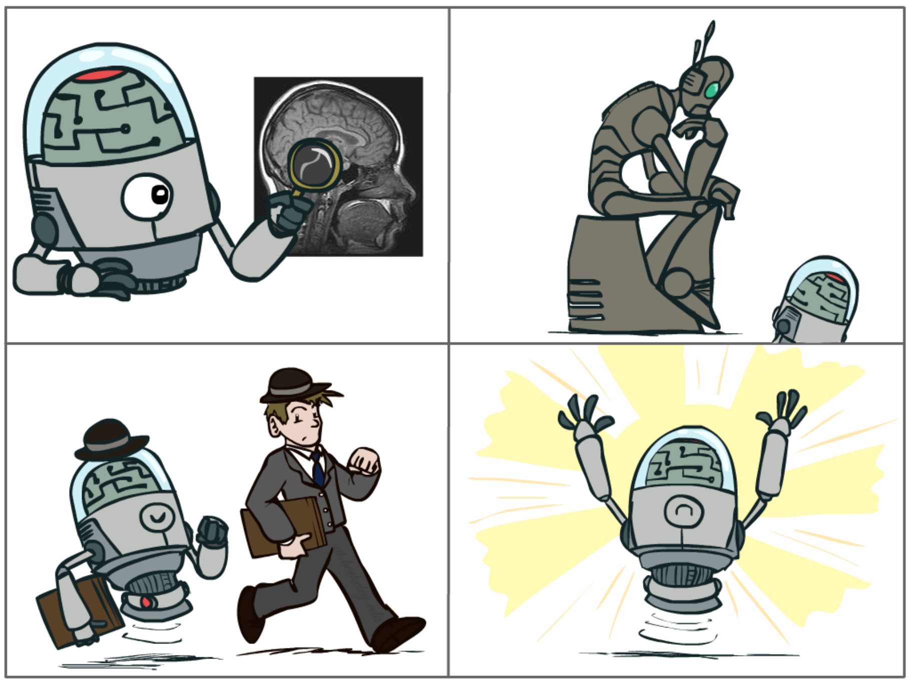
ksolaima@umbc.edu

Possible Approaches

The science of making machines that:

Think like people

Think rationally

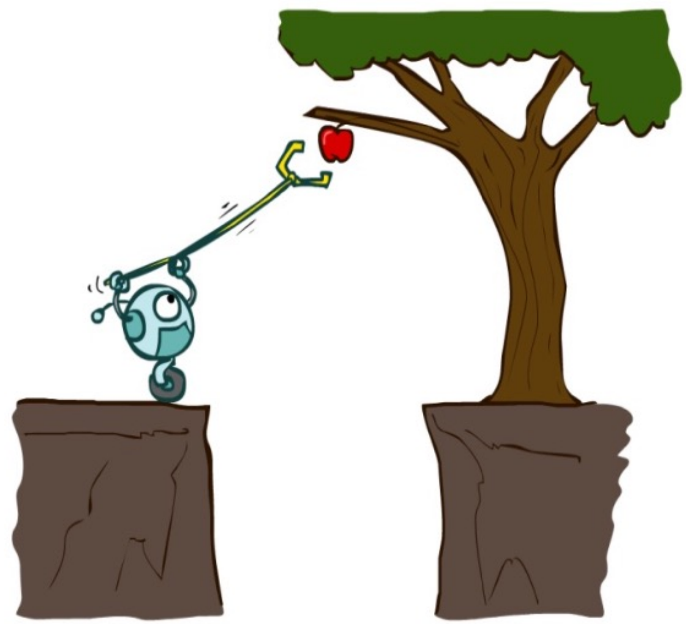


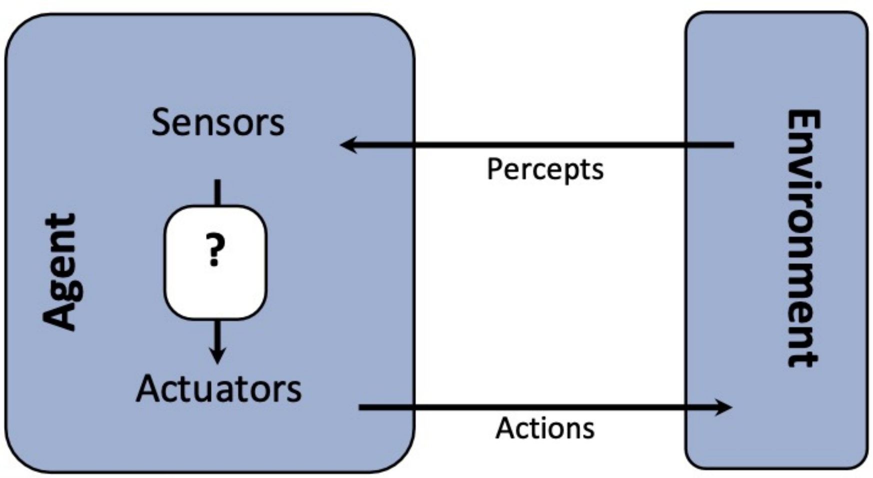
Act like people

Act rationally

Properties of an Intelligent Agent

- **Intelligent/Rational agents** select actions that maximizes its (expected) utility
- General properties an intelligent agent should have
  - **Reactive** to the environment
  - Pro-active or **goal-directed**
  - Learns/recognizes patterns
  - **Interacts** with other agents through communication or via the environment
  - **Autonomous**





Courtesy Tim Finin

(0) Table-driven agents

Use percept sequence/action table to find next action. Implemented by a **lookup table**

(1) Simple reflex agents

Based on **condition-action rules**, stateless devices with no memory of past world states

(2) Agents with memory

have **represent states** and keep track of past world states

(3) Agents with goals

Have a state and **goal information** describing desirable situations; can take future events into consideration

(4) Utility-based agents

base decisions on **utility theory** in order to act rationally

(5) Learning agents

base decisions on **models learned** and updated through experience

simple

complex

Courtesy Tim Finin

Characteristics of environments

→ Lots of real-world domains fall into the hardest case!

	Fully observable?	Deterministic?	Episodic?	Static?	Discrete?	Single agent?
Solitaire	No	Yes	Yes	Yes	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes	No
Taxi driving	No	No	No	No	No	No
Internet shopping	No	No	No	No	Yes	No
Medical diagnosis	**No**	**No**	**No**	**No**	**No**	**Yes**

A **Yes** in a cell means that aspect is simpler; a **No** more complex

Courtesy Tim Finin

A General Searching Algorithm

Core ideas:

1. Maintain a list of frontier (fringe) nodes
   1. Nodes coming *into* the frontier have been explored
   2. Nodes going out of the frontier have not been explored
2. Iteratively select nodes from the frontier and explore unexplored nodes from the frontier
3. Stop when you reach your **goal**

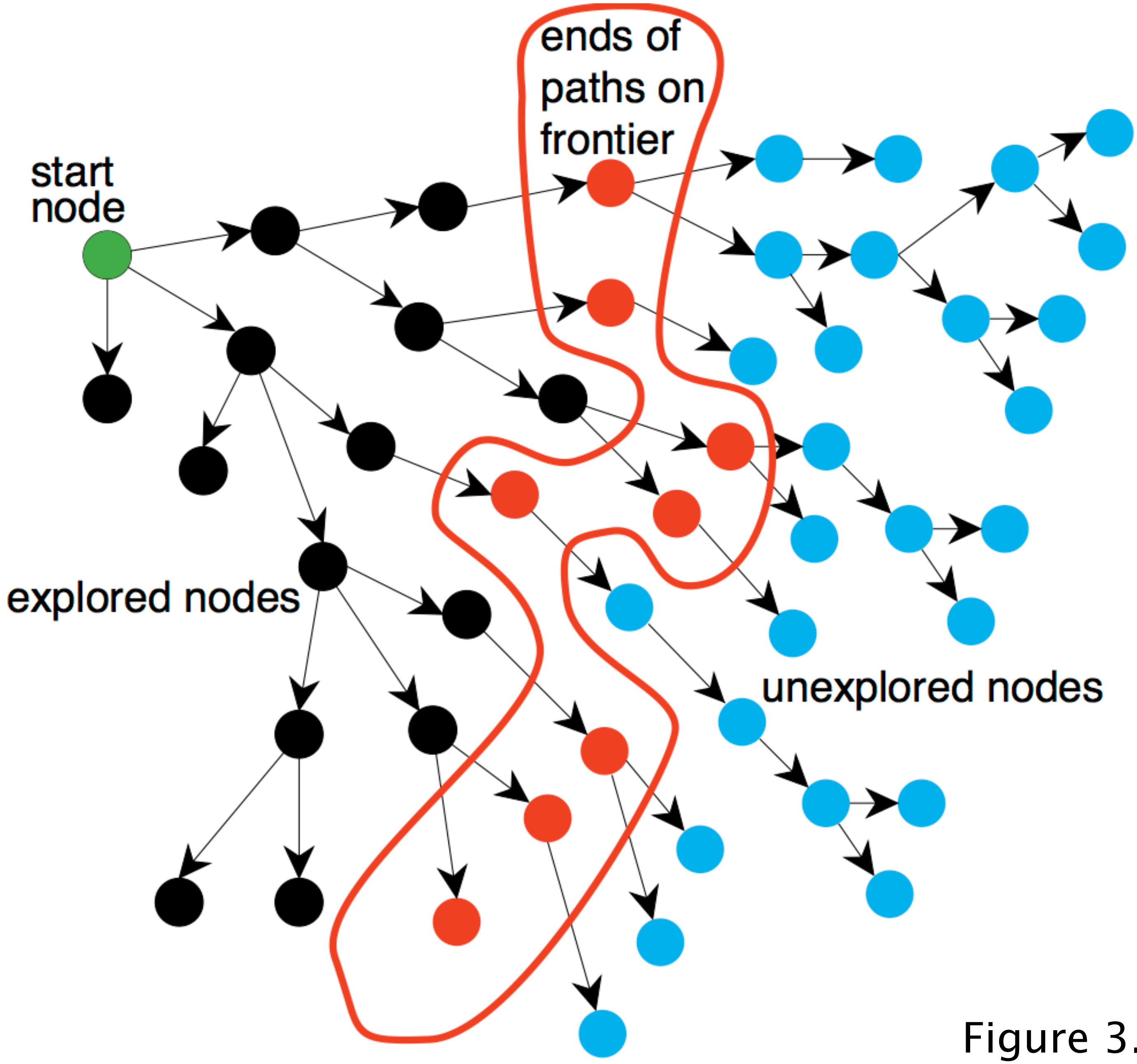
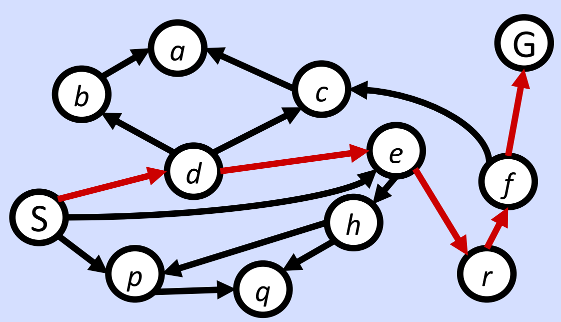


Figure 3.3

State Space Graphs vs. Search Trees

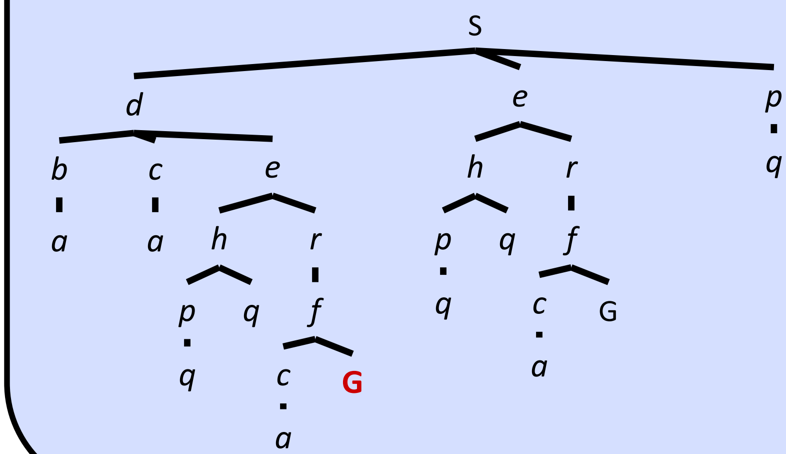
State Space Graph



Each NODE in in the search tree is an entire PATH in the state space graph.

We construct the tree on demand – and we construct as little as possible.

Search Tree



Informed vs. uninformed search

Uninformed search strategies (blind search)

– Use no information about likely *direction* of a goal
– Methods: breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional

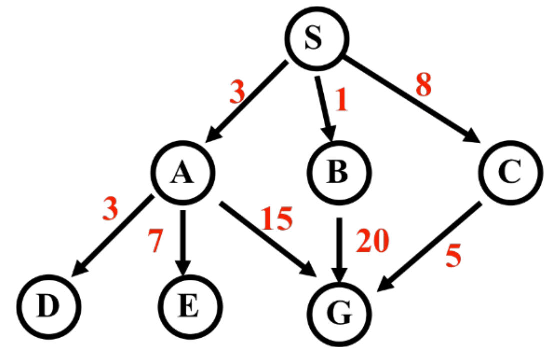
Informed search strategies (heuristic search)

– Use information about domain to (try to) (usually) head in the general direction of goal node(s)
– Methods: hill climbing, best-first, greedy search, beam search, algorithm A, algorithm A*

Evaluating search strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity** (worst or average case)
  - Usually measured by *number of nodes expanded*
- **Space complexity**
  - Usually measured by maximum size of graph/tree during the search
- **Optimality/Admissibility**
  - If a solution is found, is it **guaranteed** to be an optimal one, i.e., one with minimum cost

Uniform-Cost Search



Expanded node	Nodes list
	$\{ S^0 \}$
S^0	$\{ B^1\ A^3\ C^8 \}$
B^1	$\{ A^3\ C^8\ G^{21} \}$
A^3	$\{ D^6\ C^8\ E^{10}\ G^{18}\ G^{21} \}$
D^6	$\{ C^8\ E^{10}\ G^{18}\ G^{21} \}$
C^8	$\{ E^{10}\ G^{13}\ G^{18}\ G^{21} \}$
E^{10}	$\{ G^{13}\ G^{18}\ G^{21} \}$
G^{13}	$\{ G^{18}\ G^{21} \}$

priority queue

Solution path found is S C G, cost 13

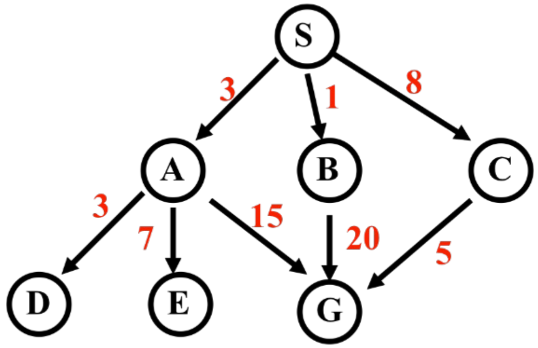
Number of nodes expanded (including goal node) = 7

Depth-First Iterative Deepening (DFID)

- Do DFS to depth 0, then (if no solution) DFS to depth 1, etc.
- Usually used with a tree search
- **Complete**
- **Optimal/Admissible** if all operators have unit cost, else finds shortest solution (like BFS)
- Time complexity a bit worse than BFS or DFS

  Nodes near top of search tree generated many times, but since almost all nodes are near tree bottom, worst case time complexity still exponential, $O(b^d)$

How they perform



- **Depth-First Search:**
  - 4 Expanded nodes: S A D E G
  - Solution found: S A G (cost 18)
- **Breadth-First Search**:
  - 7 Expanded nodes: S A B C D E G
  - Solution found: S A G (cost 18)
- **Uniform-Cost Search**:
  - 7 Expanded nodes: S A D B C E G
  - Solution found: S C G (cost 13)

  Only uninformed search that worries about costs
- **Iterative-Deepening Search**:
  - 10 nodes expanded: S S A B C S A D E G
  - Solution found: S A G (cost 18)

Comparing Search Strategies

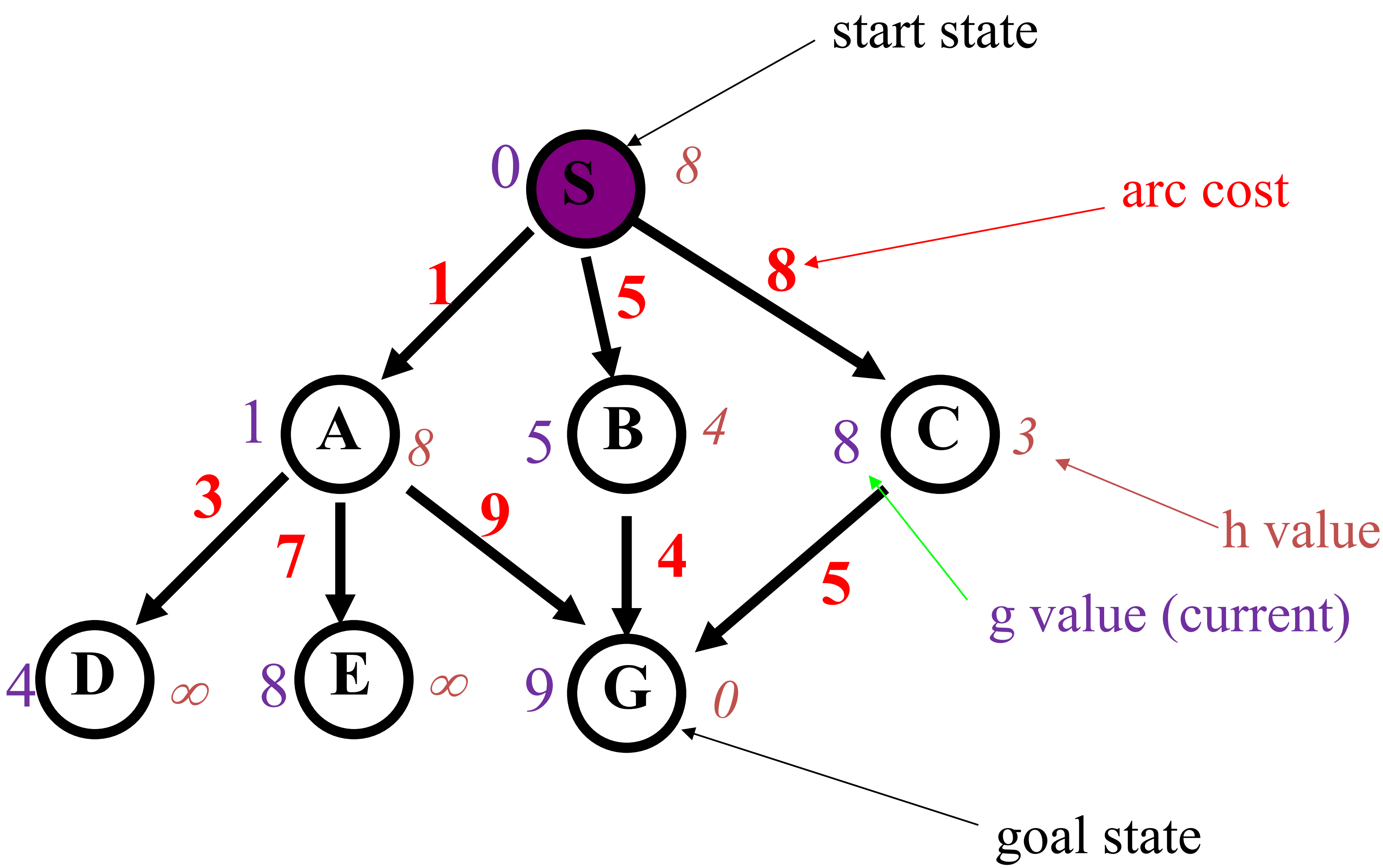
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

A* Search

Use an evaluation function

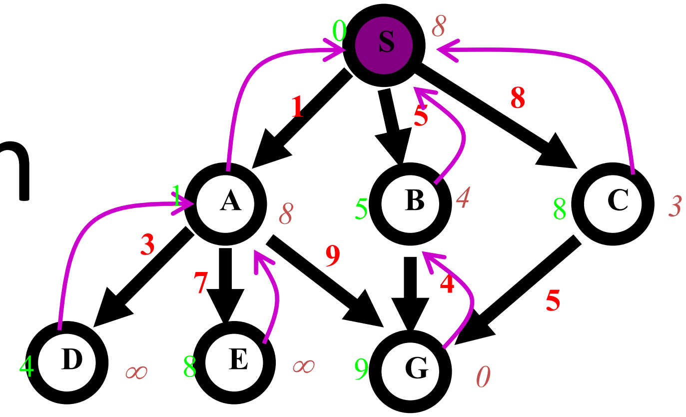
$$f(n) = g(n) + h(n)$$

estimated total cost from start to goal via state n = minimal-cost path from the start state to state n + cost estimate from state n to the goal


start state
0
S
8
arc cost
1
5
8
1
A
8
5
B
4
8
C
3
3
7
9
4
5
h value
g value (current)
4
D
∞
8
E
∞
9
G
0
goal state

GREEDY VS A*

Greedy search



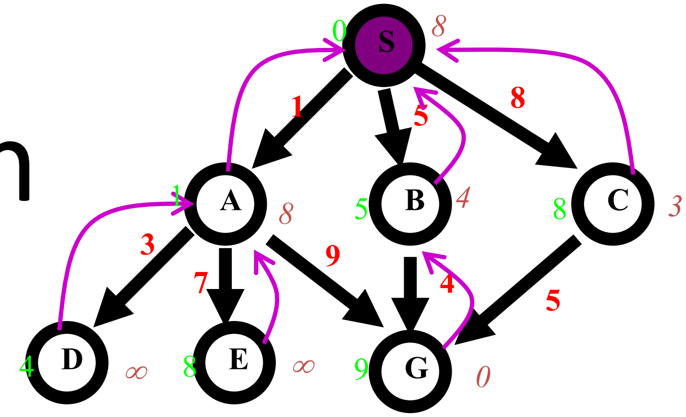
f(n) = h(n)

node expanded **nodes list**

{ S(8) }

what’s next???

Greedy search

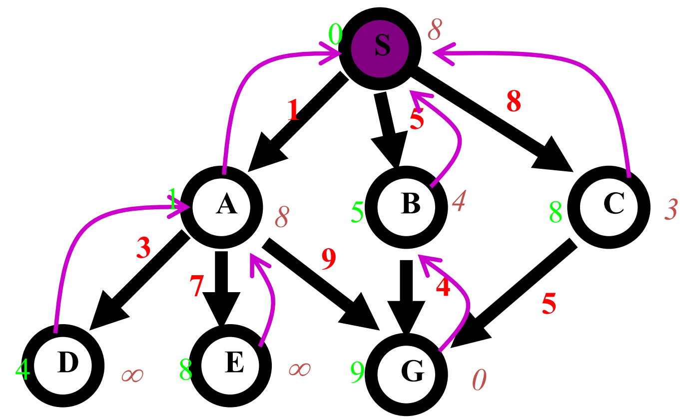


$f(n) = h(n)$

node expanded	**nodes list**
	{ S(8) }
S	{ C(3) B(4) A(8) }
C	{ G(0) B(4) A(8) }
G	{ B(4) A(8) }

- Solution path found is S C G, 3 nodes expanded.
- See how fast the search is!! But it is NOT optimal.

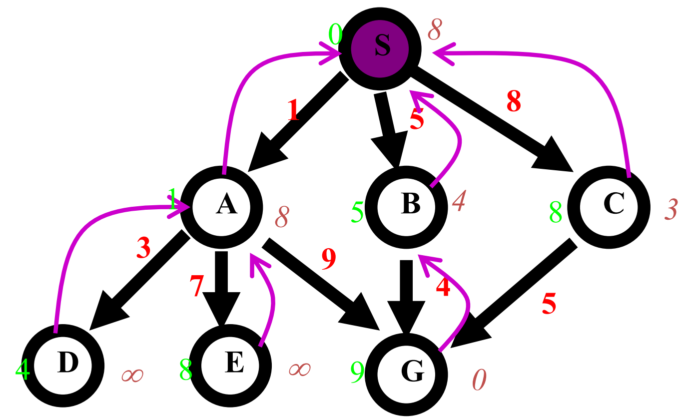
A* search



f(n) = g(n) + h(n)

```
node exp.      nodes list
            { S(8) }
                       What's next?
```

A* search



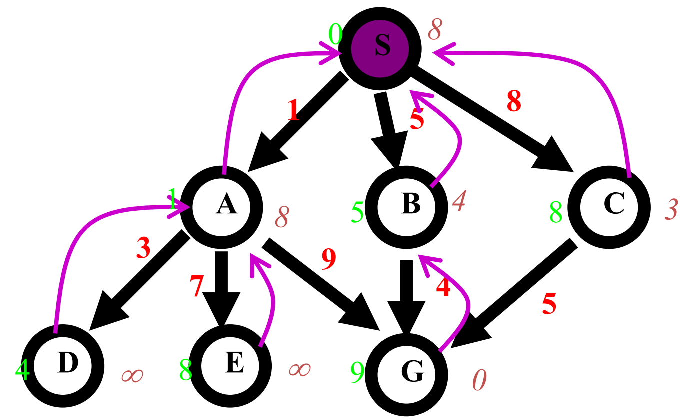
f(n) = g(n) + h(n)

```
node exp.        nodes list
             { S(8) }
 S           { A(9) B(9) C(11) }
                 What's next?
```

h(n)

h(S)=8
h(A)=8
h(B)=4
h(C)=3
h(D)=inf
h(E)=inf
h(G)=0

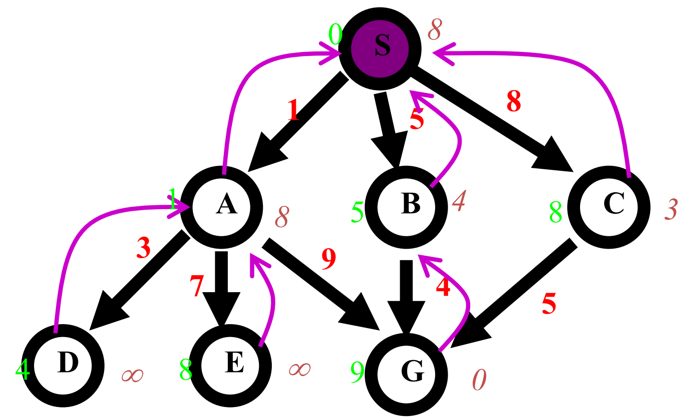
A* search



f(n) = g(n) + h(n)

```
node exp.        nodes list
                 { S(8) }
 S               { A(9) B(9) C(11) }
 A               { B(9) G(10) C(11) D(inf) E(inf) }
                      What's next?
```

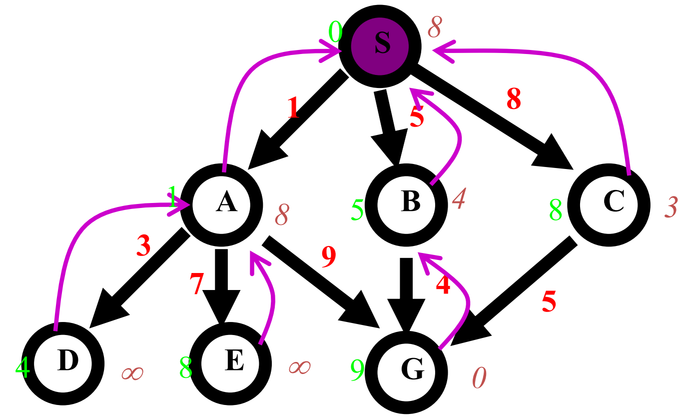
A* search



f(n) = g(n) + h(n)

```
node exp.        nodes list
              { S(8) }
 S            { A(9) B(9) C(11) }
 A            { B(9) G(10) C(11) D(inf) E(inf) }
 B            { G(9) G(10) C(11) D(inf) E(inf) }
                     What's next?
```

A* search



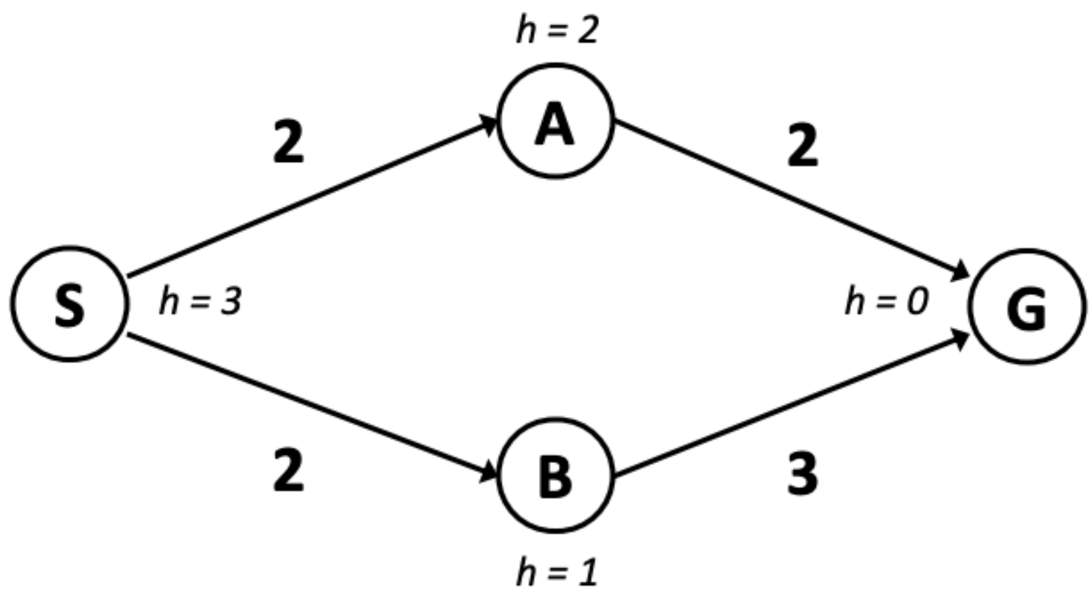
f(n) = g(n) + h(n)

```
node exp.      nodes list
             { S(8) }
 S           { A(9) B(9) C(11) }
 A           { B(9) G(10) C(11) D(inf) E(inf) }
 B           { G(9) G(10) C(11) D(inf) E(inf) }
 G           { C(11) D(inf) E(inf) }
```

- Solution path found is S B G, 4 nodes expanded..
- Still pretty fast. And optimal, too.

When should A* terminate?

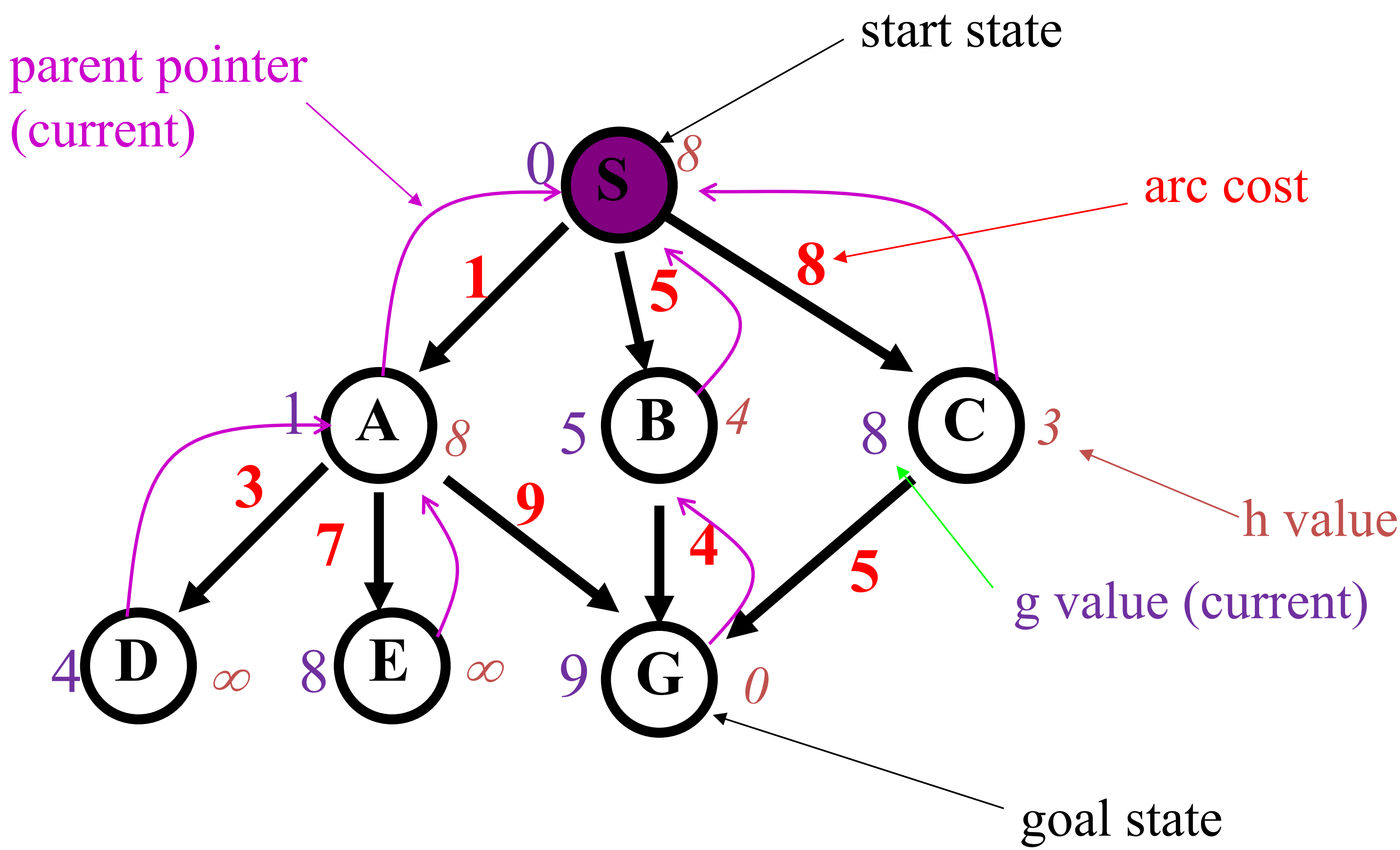
- Should we stop when we enqueue a goal?



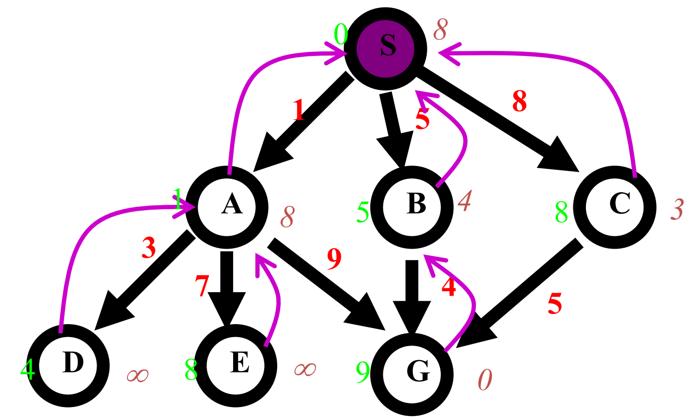
- No: only stop when we dequeue a goal

IS A HEURISTIC ADMISSIBLE?

Example search space



Example

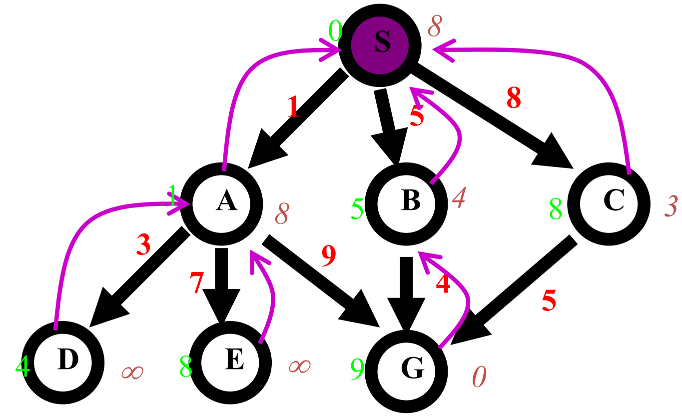


n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	9

- h*(n) is (hypothetical) perfect heuristic (an oracle)
- Since h(n) <= h*(n) for all n, h is admissible (optimal)
- Optimal path = *S B G* with cost 9

The table and graph show values for the entire space, but we must discover or compute them during the search

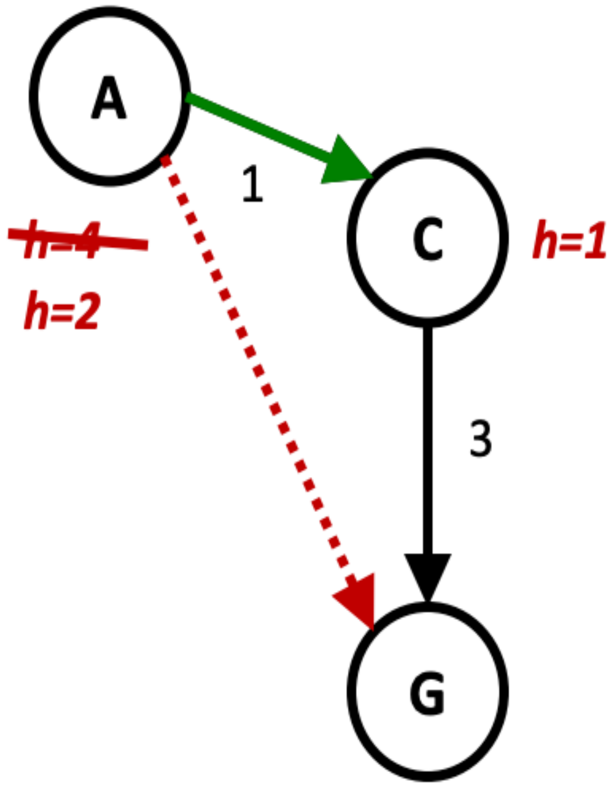
Example



n	**g(n)**	**h(n)**	**f(n)**	**h*(n)**
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	inf	inf	inf
E	8	inf	inf	inf
G	9	0	9	0

- h*(n) is (hypothetical) perfect heuristic (an oracle)
- **Since h(n) <= h*(n) for all n, h is admissible (optimal)**
- Optimal path = *S B G* with cost 9

Consistency of Heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G

  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

    h(A) – h(C) ≤ cost(A to C)

- Consequences of consistency:
  - The f value along a path never decreases

    h(A) ≤ cost(A to C) + h(C)

  - A* graph search is optimal

Hill-climbing search

- If there's successor **s** for current state **n** such that
  - $h(s) < h(n)$ and $h(s) <= h(t)$ for all successors t

  then move from **n** to **s**; otherwise, halt at **n**

  i.e.: Look one step ahead to decide if a successor is better than current state; if so, move to best successor
- Like *greedy search*, but doesn't allow backtracking or jumping to alternative path since it has no memory
- Like beam search with a beam width of 1 (i.e., maximum size of the nodes list is 1)
- Not complete since search may terminate at a local minima, plateau or ridge

Drawbacks of hill climbing

- Problems: local maxima, plateaus, ridges
- Possible remedies:
  - **Random restart:** keep restarting search from random locations until a goal is found

    may require an estimate – *how low can we go*
  - **Problem reformulation:** reformulate search space to eliminate these problematic features
- Some problem spaces are great for hill climbing and others are terrible

SA intuitions

- Combines **hill climbing** (for efficiency) with **random walk** (for completeness)
- Analogy: get ping-pong ball into the deepest depression in bumpy surface
  - Shake surface to get the ball out of local minima
  - Don’t shake too hard to dislodge it from global minimum
- Simulated annealing:
  - Start shaking hard (high temperature) and gradually reduce shaking intensity (lower temperature)
  - Escape local minima by allowing some “bad” moves
  - But gradually reduce their size and frequency

Simulated annealing

- "bad" move from A to B accepted with prob.

  $$e^{-(f(B)-f(A)/T)}$$

- The higher the temperature, the more likely it is that a bad move can be made
- As T tends to zero, probability tends to zero, and SA becomes more like hill climbing
- If T lowered slowly enough, SA is complete and admissible
- Finding proper rate to lower still an issue

Genetic algorithms

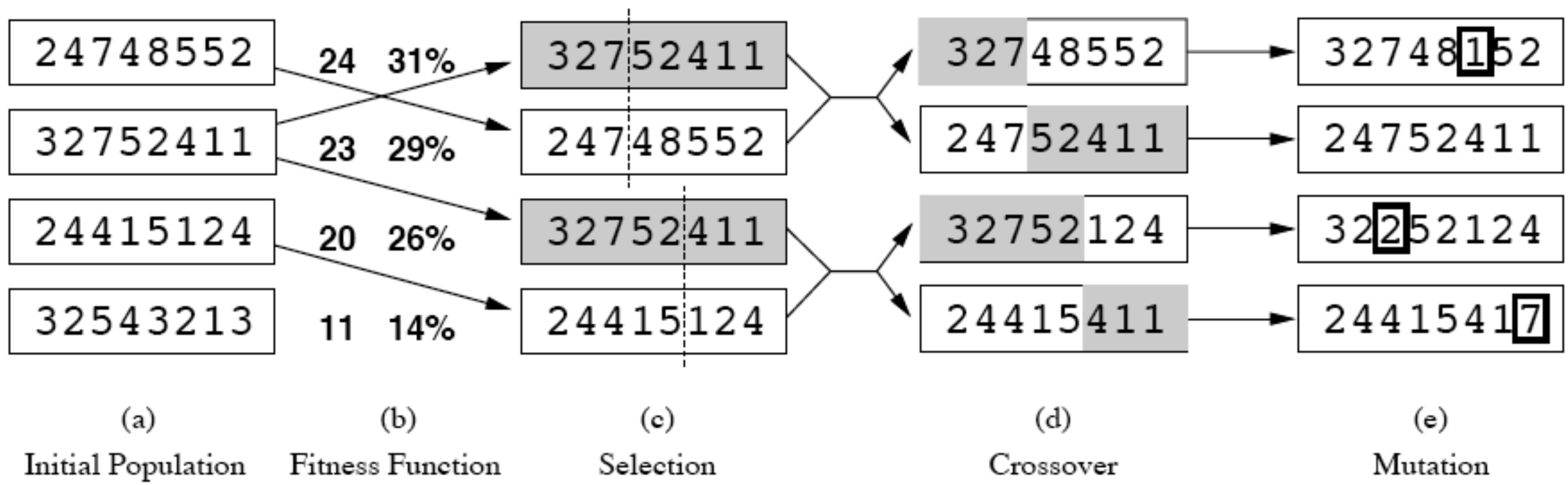


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- **Fitness function**: number of non-attacking pairs of queens (min=0, max=(8 × 7)/2 = 28)
- **Probability of mating** is a function of fitness score
- **Random cross-over point** for a mating pair chosen
- Resulting offspring subject to a **random mutation with probability**

Selection

- Random, or
- Roulette wheel Selection
  - Fitness Function
  - Take % of fitness score
  - Higher the fitness score, higher the %, higher the chance of getting selected
  - Fitness proportionate selection
  - 14% is never selected, 31% is selected twice

Genetic algorithms

Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

- **Fitness function**: number of non-attacking pairs of queens (min=0, max=(8 × 7)/2 = 28)
- **Probability of mating** is a function of fitness score
- **Random cross-over point** for a mating pair chosen
- Resulting offspring subject to a **random mutation with probability**

Genetic algorithms

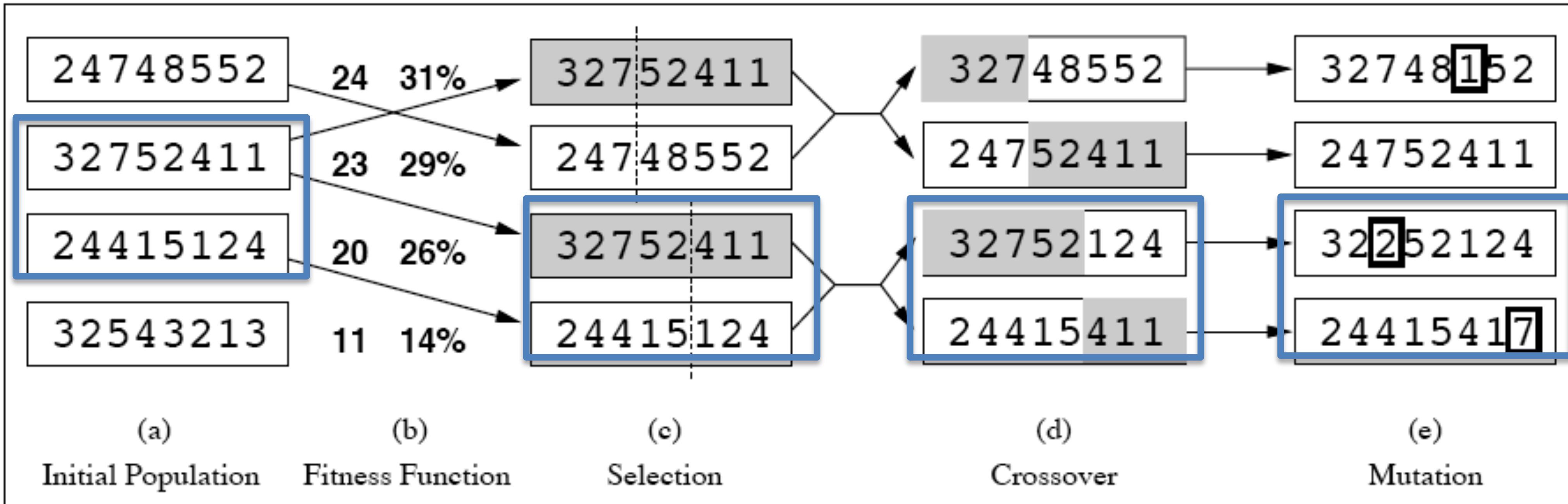
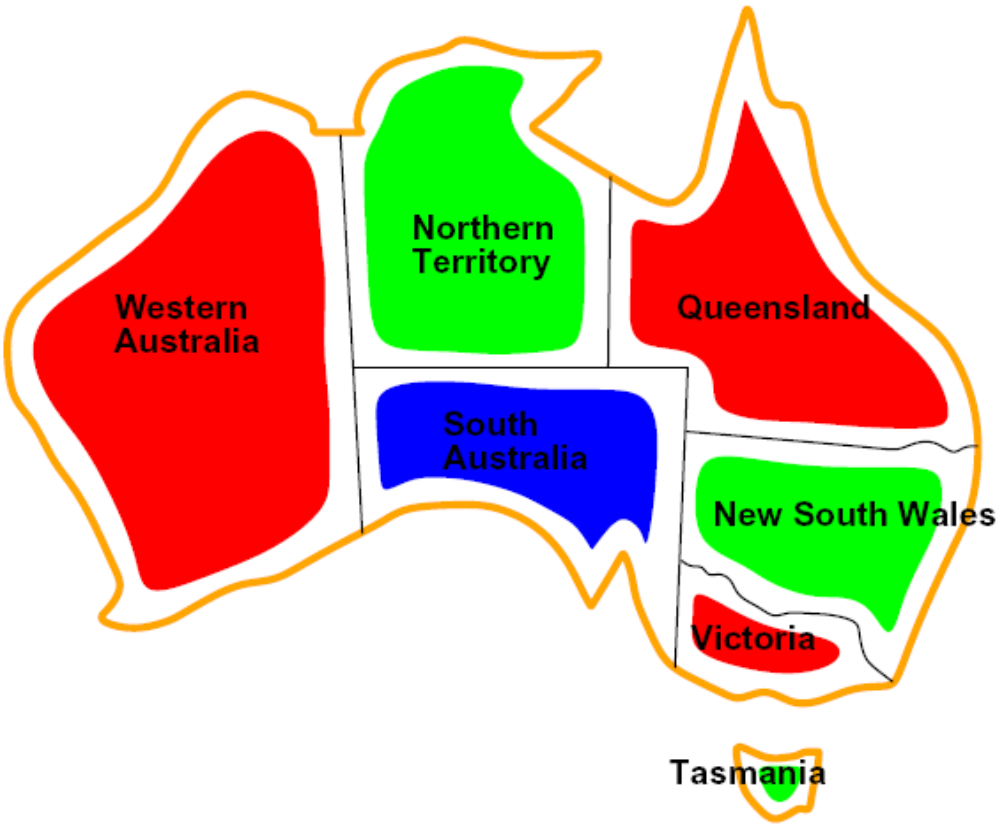


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

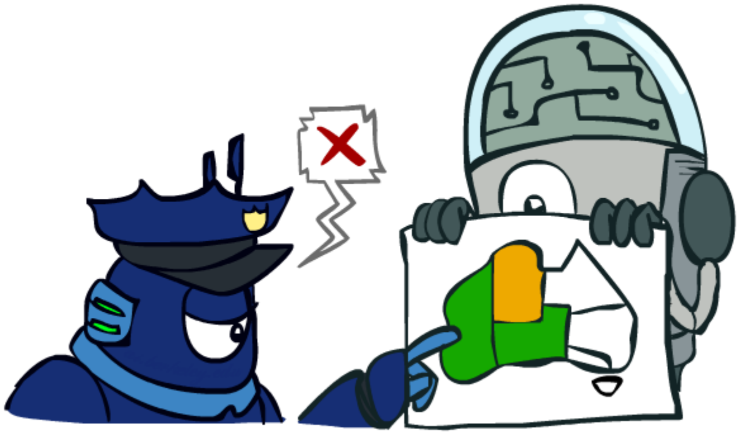
- **Fitness function**: number of non-attacking pairs of queens (min=0, max=(8 × 7)/2 = 28)
- **Probability of mating** is a function of fitness score
- **Random cross-over point** for a mating pair chosen
- Resulting offspring subject to a **random mutation with probability**

CSP Examples

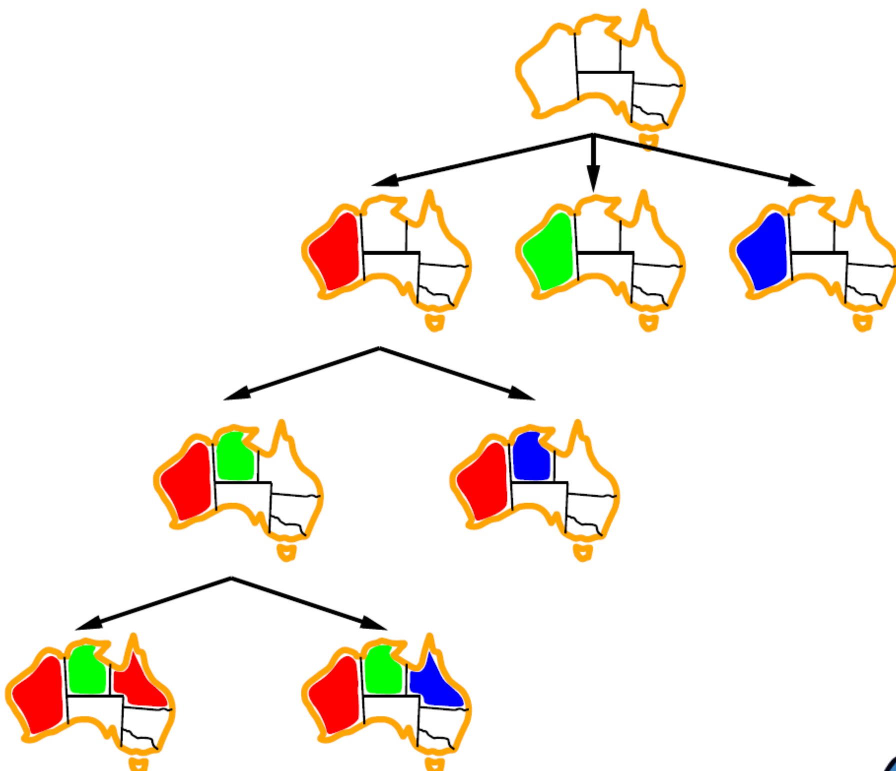


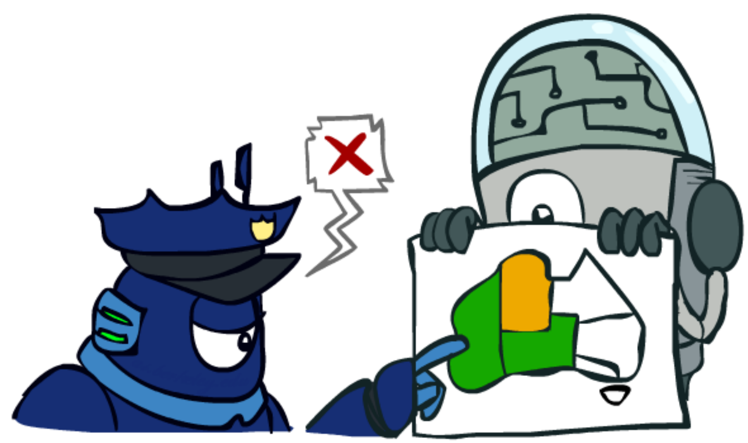
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name)



Backtracking Example





Enforcing Arc Consistency in a CSP

function AC-3(*csp*) **returns** the CSP, possibly with reduced domains
  inputs: *csp*, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) **then**
      for each X_k **in** NEIGHBORS[X_i] **do**
        add (X_k, X_i) to *queue*

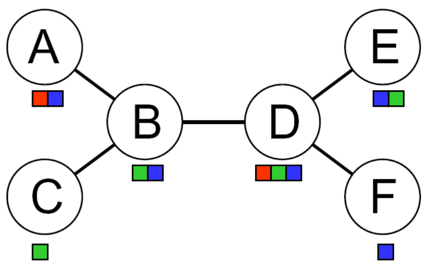
---

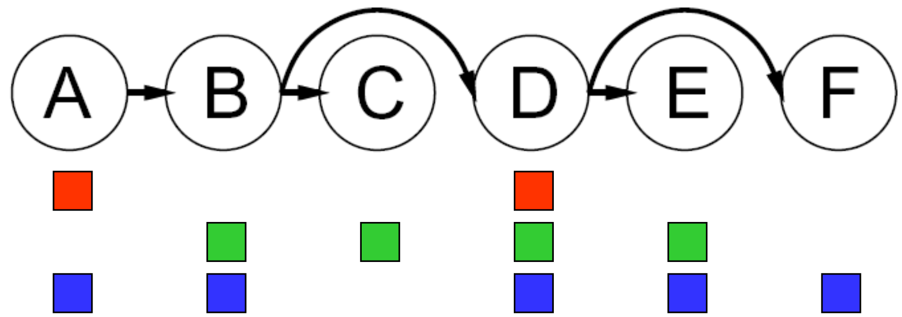
function REMOVE-INCONSISTENT-VALUES(X_i, X_j) **returns** true iff succeeds
  removed ← *false*
  for each x **in** DOMAIN[X_i] **do**
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint $X_i \leftrightarrow X_j$
      then delete x from DOMAIN[X_i]; *removed* ← *true*
  return *removed*

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

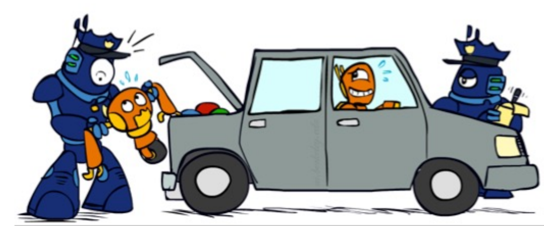
Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children





  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)



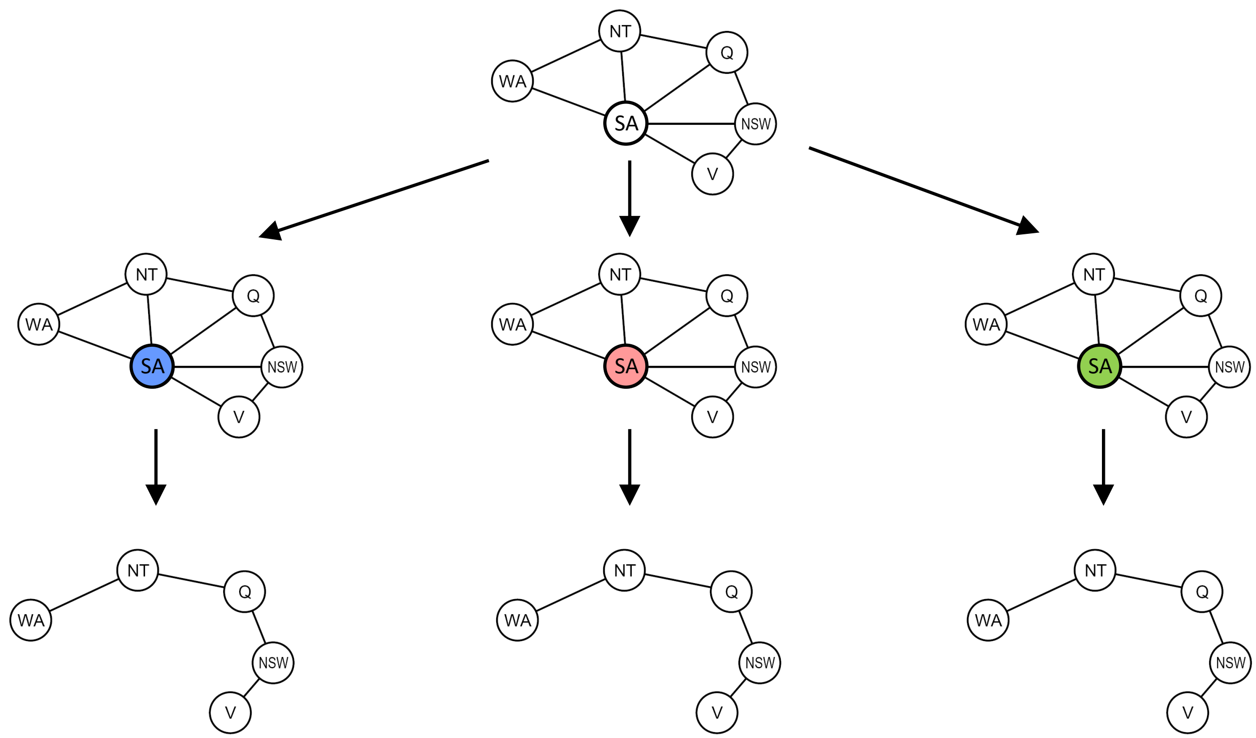
Cutset Conditioning

Choose a cutset

Instantiate the cutset
(all possible ways)

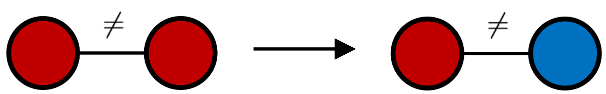
Compute residual CSP
for each assignment

Solve the residual CSPs
(tree structured)



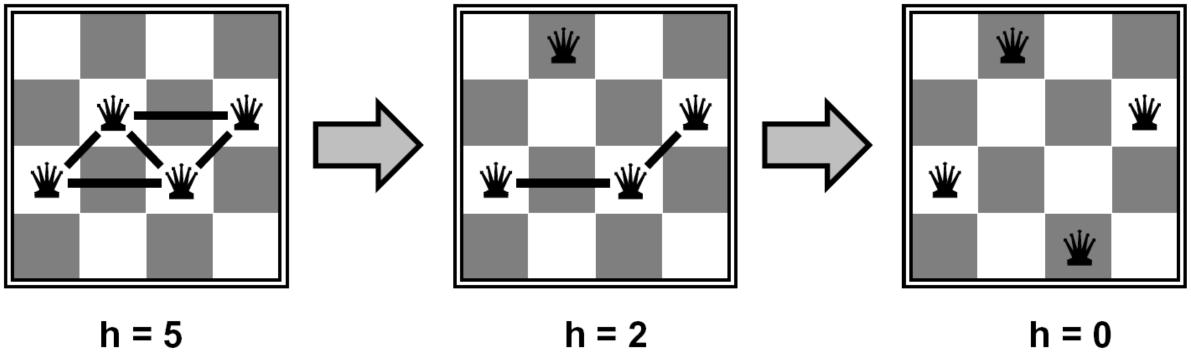
Iterative Algorithms for CSPs

- Local search methods typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.



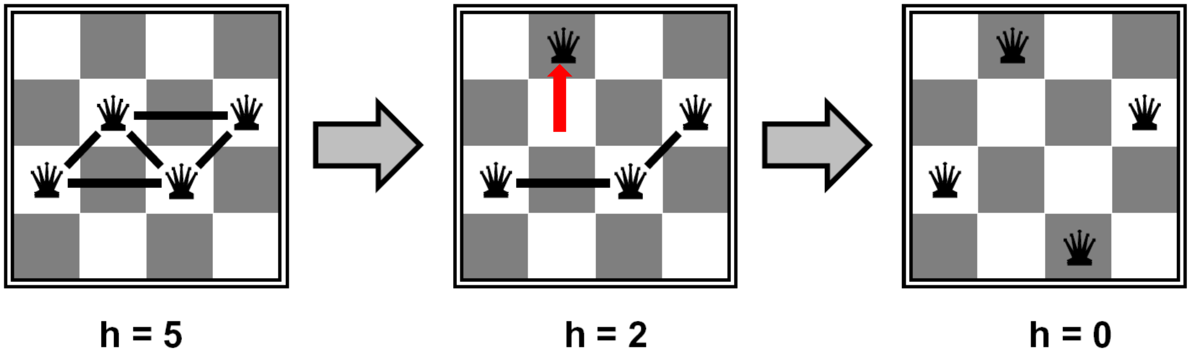
- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - I.e., hill climb with h(n) = total number of violated constraints

Example: 4-Queens



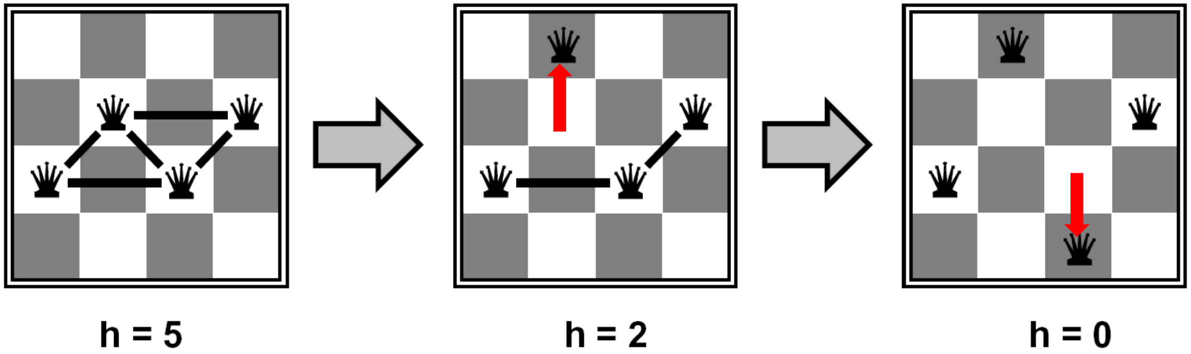
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Example: 4-Queens



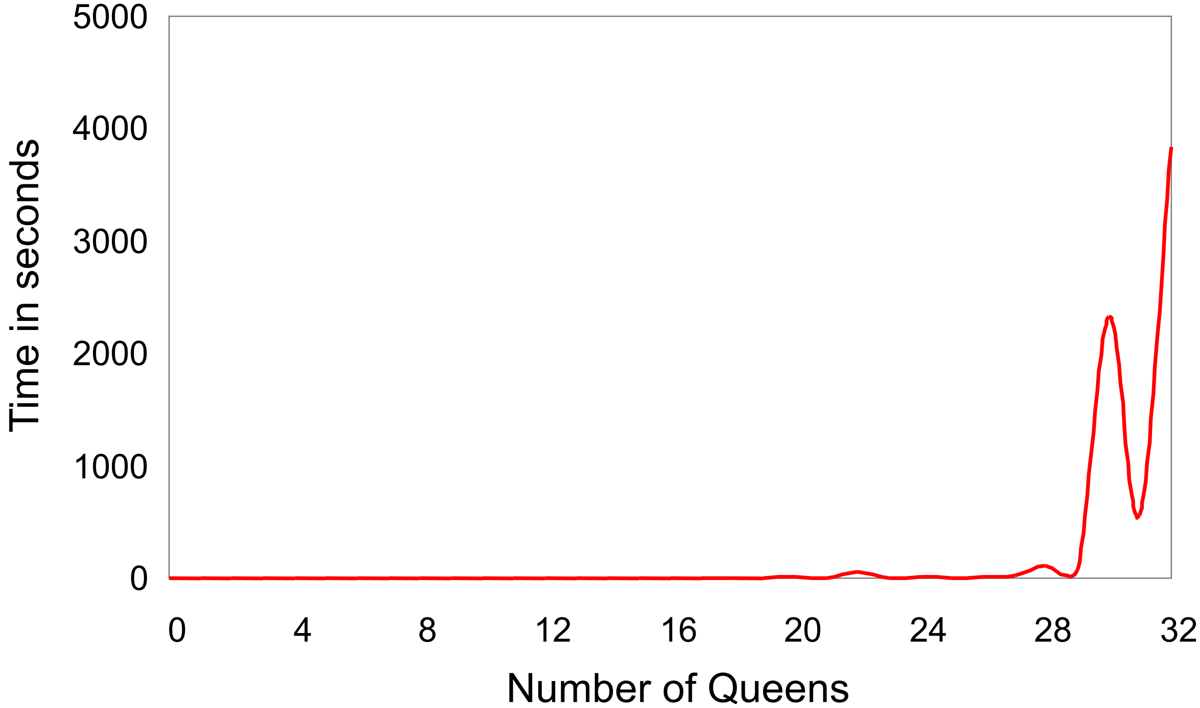
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Example: 4-Queens



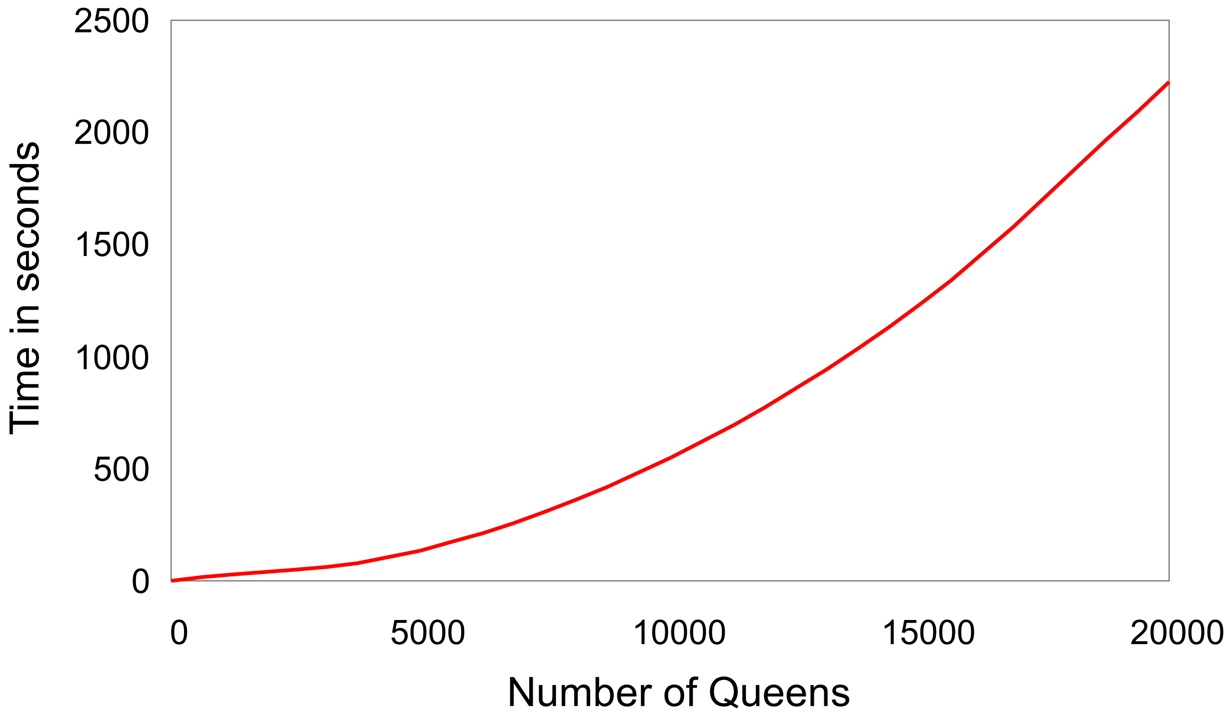
- States: 4 queens in 4 columns (4^4 = 256 states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Backtracking Performance



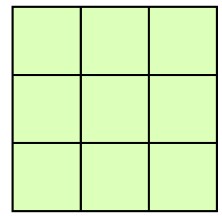
Slide

Local Search Performance

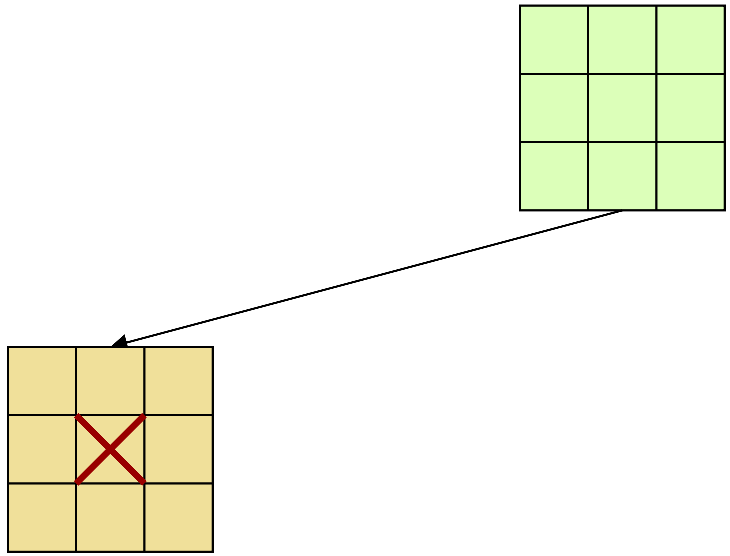


Slide

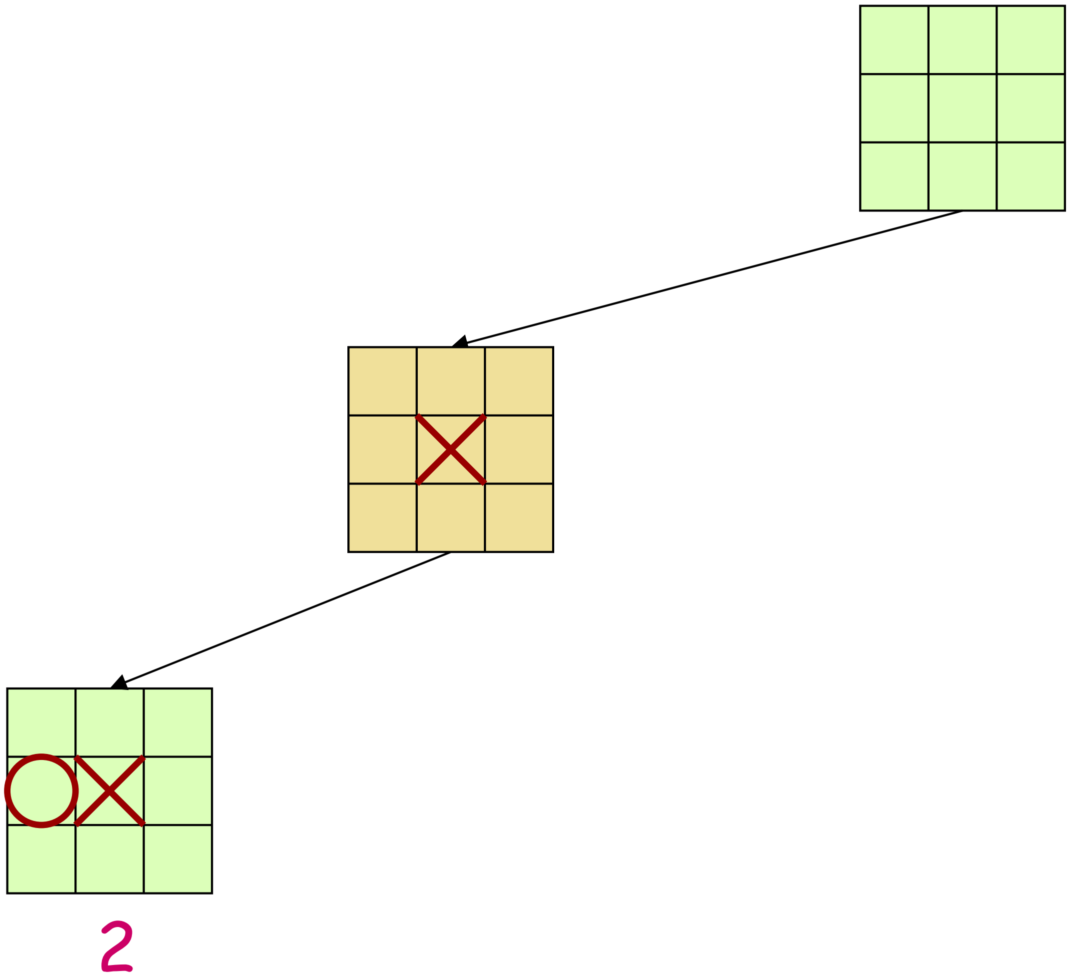
Alpha-Beta Tic-Tac-Toe Example



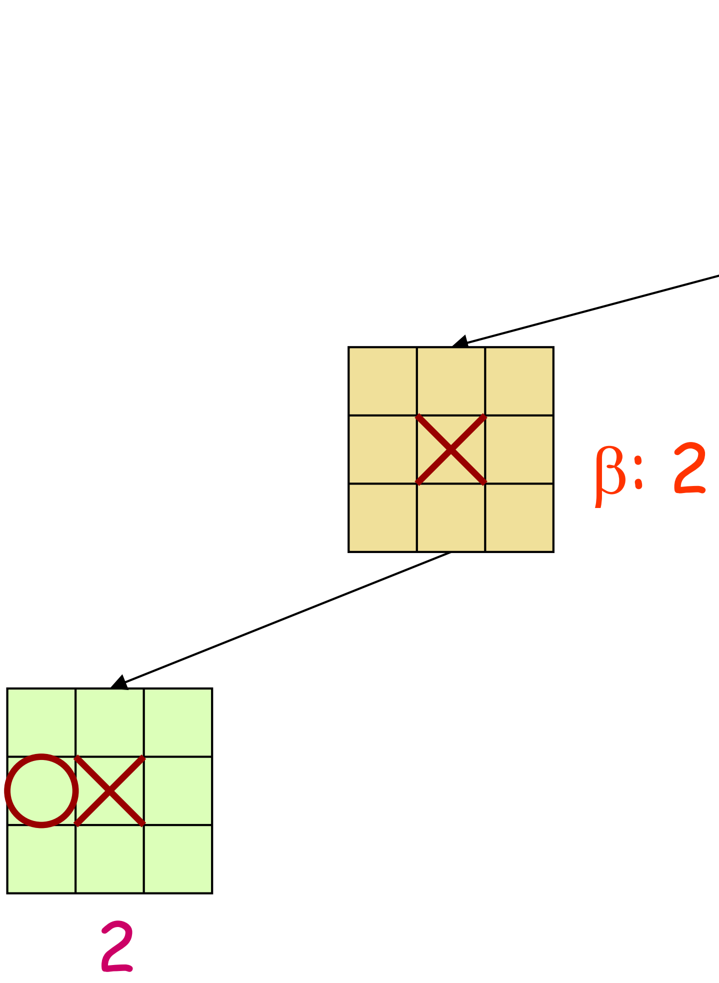
Alpha-Beta Tic-Tac-Toe Example



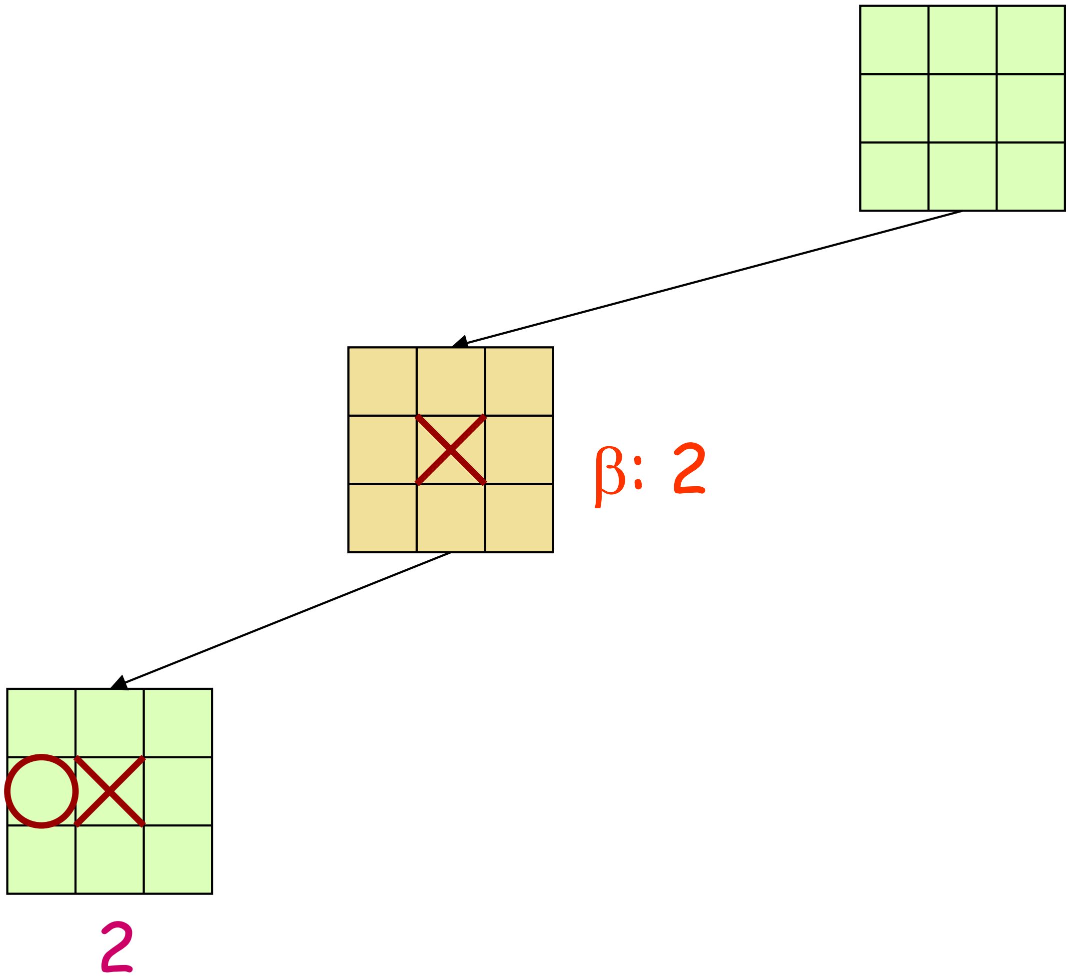
Alpha-Beta Tic-Tac-Toe Example



Alpha-Beta Tic-Tac-Toe Example

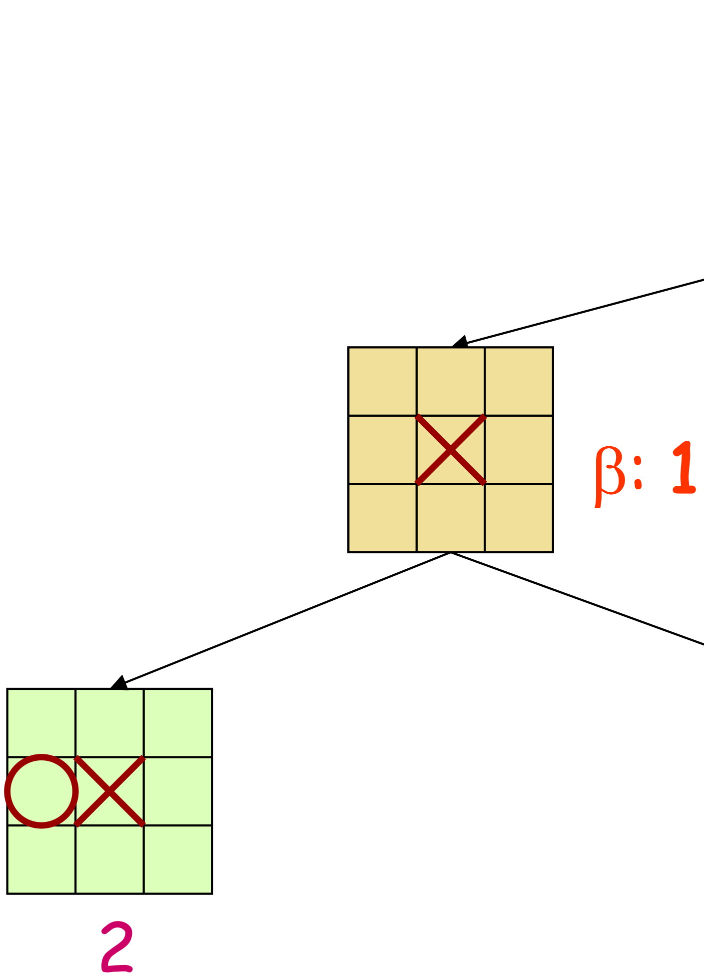


Alpha-Beta Tic-Tac-Toe Example

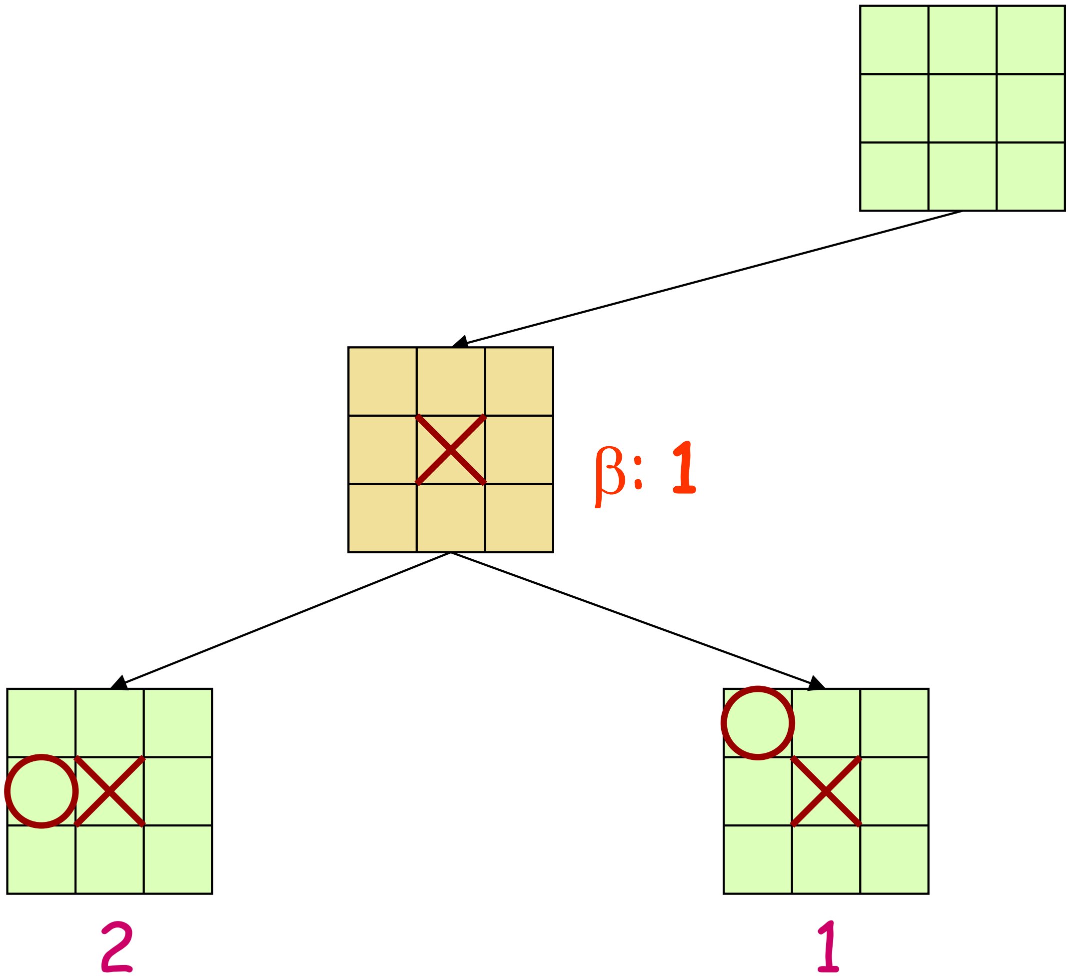


Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

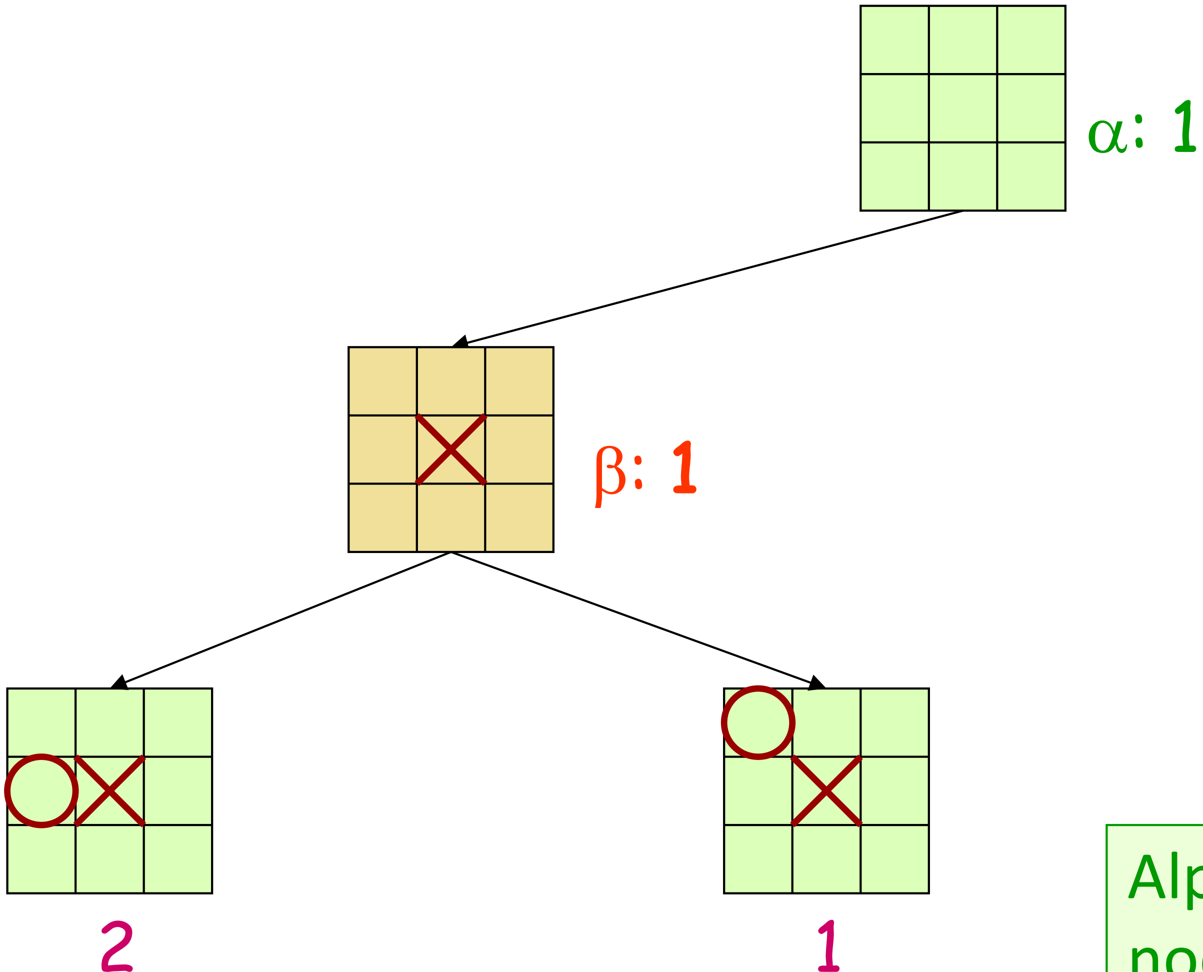


Alpha-Beta Tic-Tac-Toe Example



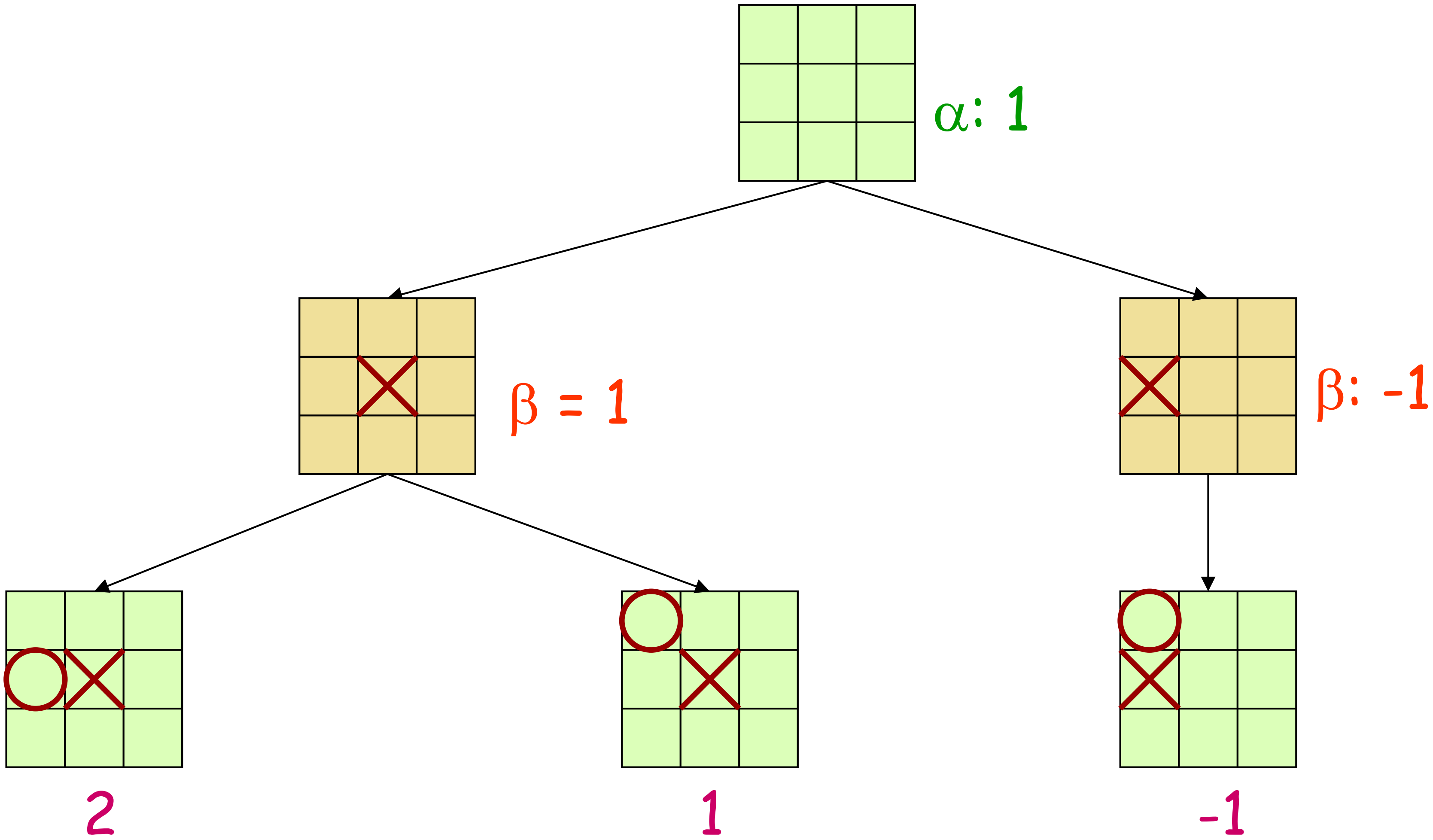
Beta value of a MIN node is **upper** bound on final backed-up value; it can never increase

Alpha-Beta Tic-Tac-Toe Example

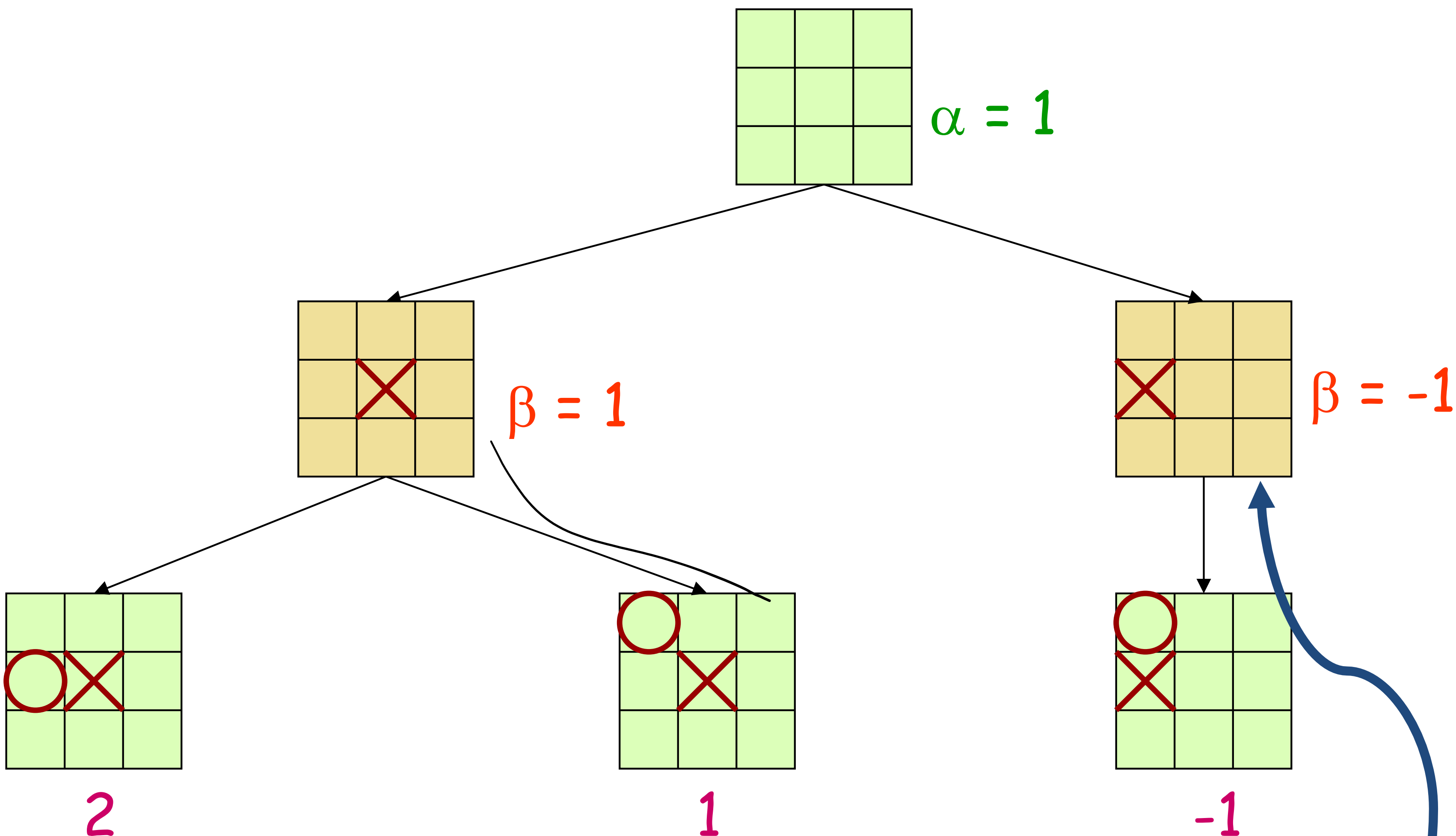


Alpha value of MAX node is **lower** bound on final backed-up value; it can never decrease

Alpha-Beta Tic-Tac-Toe Example

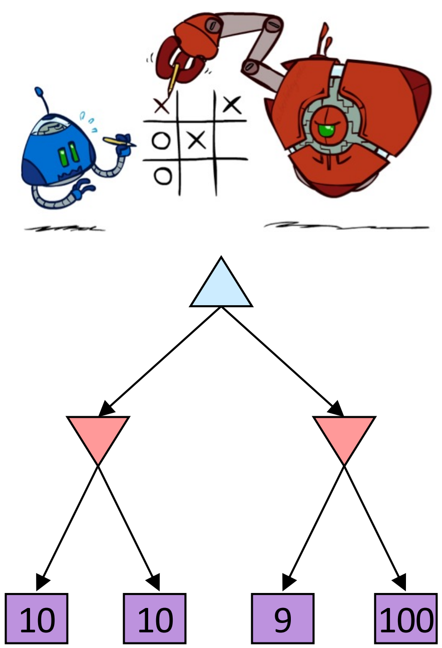


Alpha-Beta Tic-Tac-Toe Example

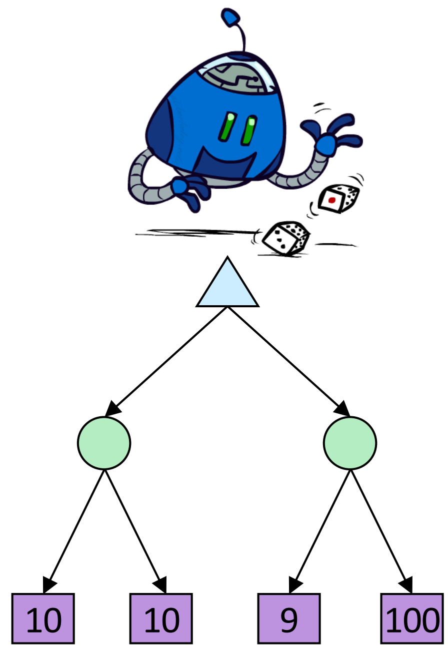


Discontinue search below a MIN node whose beta value ≤ alpha value of one of its MAX ancestors

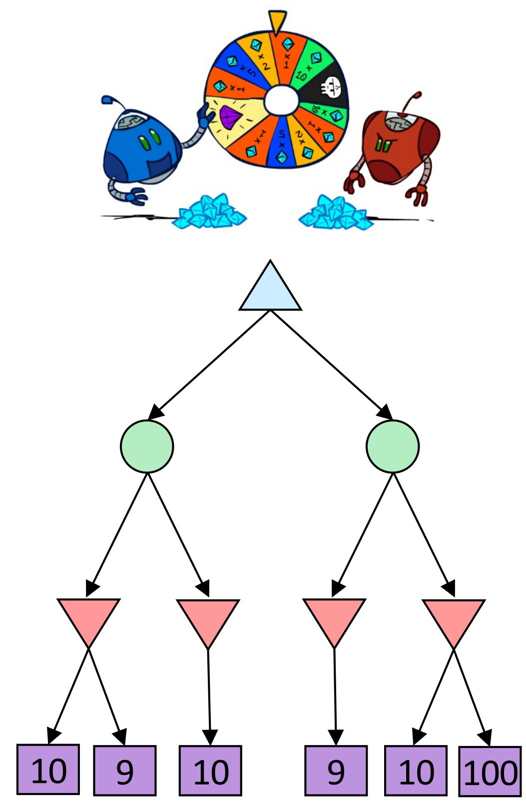
Chance outcomes in trees



Tictactoe, chess
Minimax



Tetris, investing
Expectimax

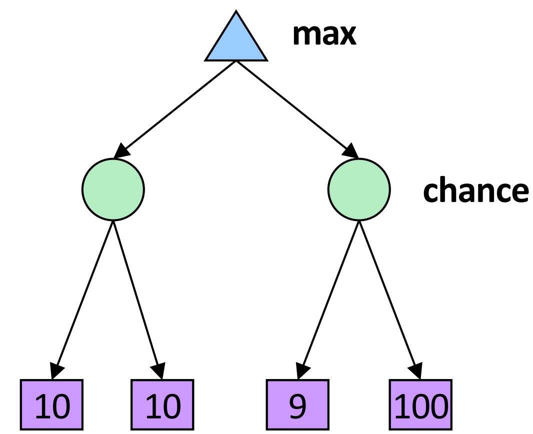


Backgammon, Monopoly
Expectiminimax

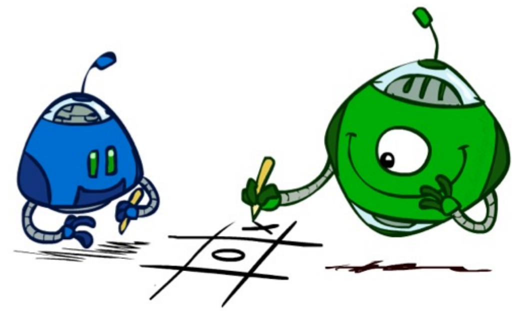
Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

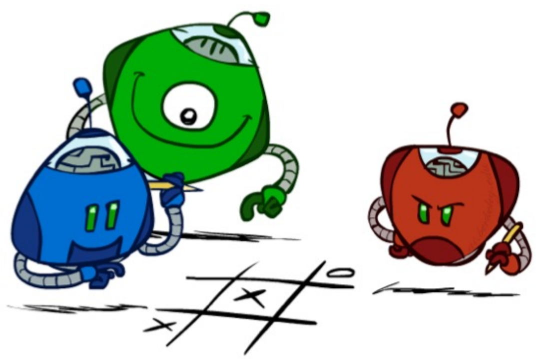
- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

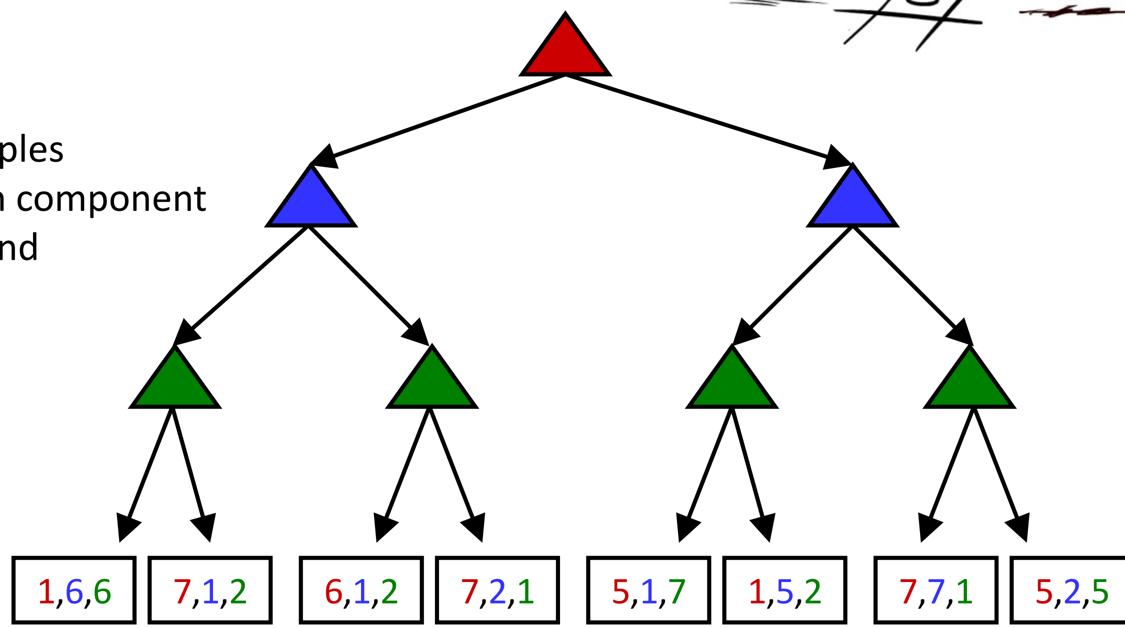


Multi-Agent Utilities



- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…





Monte Carlo Tree Search

- Methods based on alpha-beta search assume a fixed horizon
  - Pretty hopeless for Go, with $b > 300$
- MCTS combines two important ideas:
  - ***Evaluation by rollouts*** – play multiple games to termination from a state s (using a simple, fast rollout policy) and count wins and losses
  - ***Selective search*** – explore parts of the tree that will help improve the decision at the root, regardless of depth

Upper Confidence Bounds (UCB) heuristics

- UCB1 formula combines "promising" and "uncertain":
  - C is a parameter we choose to trade off between two terms

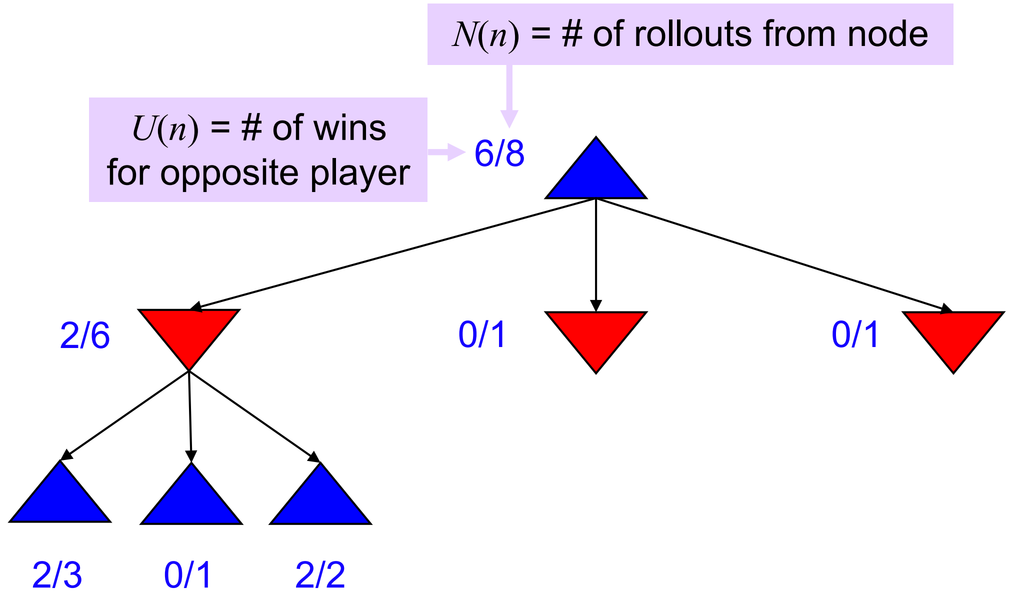
$$UCB1(n) = \frac{U(n)}{N(n)} + \boxed{C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}}$$

- High for small N
- Low for large N

- $N(n)$ = number of rollouts from node *n*
- $U(n)$ = total utility of rollouts (# wins) for player of Parent(*n*)
  - Keep track of both N and U for each node

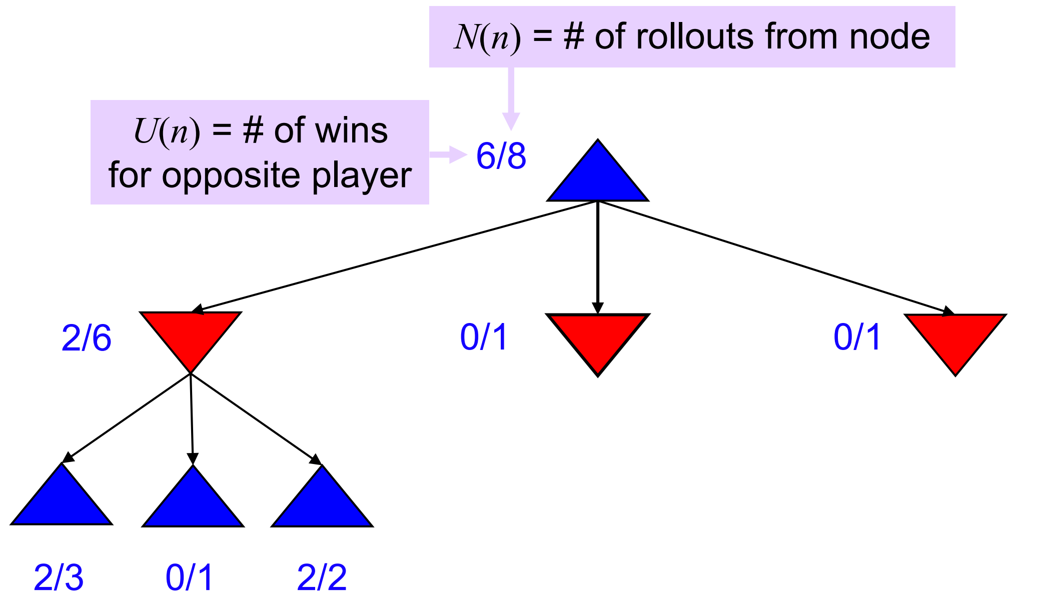
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



MCTS Algorithm

- Repeat until out of time:
  - **Selection**: recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



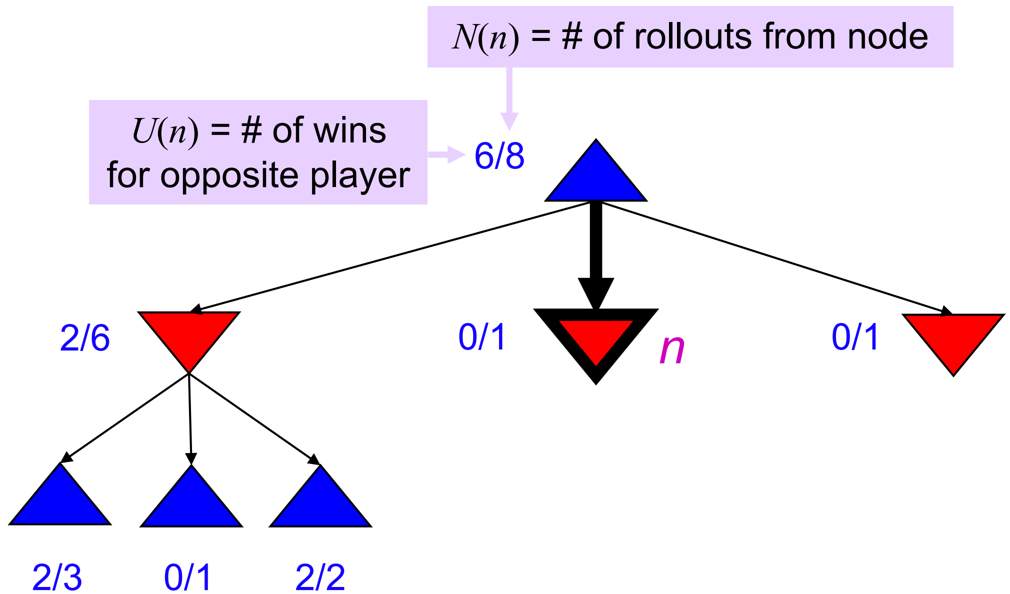
For 3 red nodes above the UCB values (with C=1) are:

$$\frac{2}{6} + \sqrt{\frac{\log 8}{6}} \quad 1.06 \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}} \quad 1.75 \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}}$$

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}$$

MCTS Algorithm

- Repeat until out of time:
  - **Selection**: recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



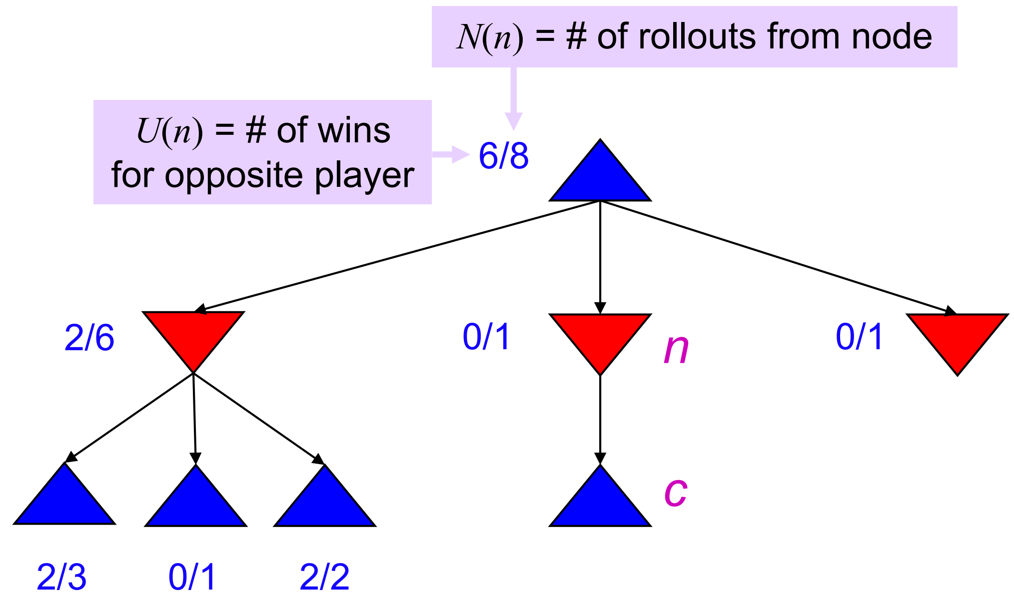
$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{Parent}(n))}{N(n)}}$$

For 3 red nodes above the UCB values (with C=1) are:

$$\frac{2}{6} + \sqrt{\frac{\log 8}{6}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}} \qquad \frac{0}{1} + \sqrt{\frac{\log 8}{1}}$$

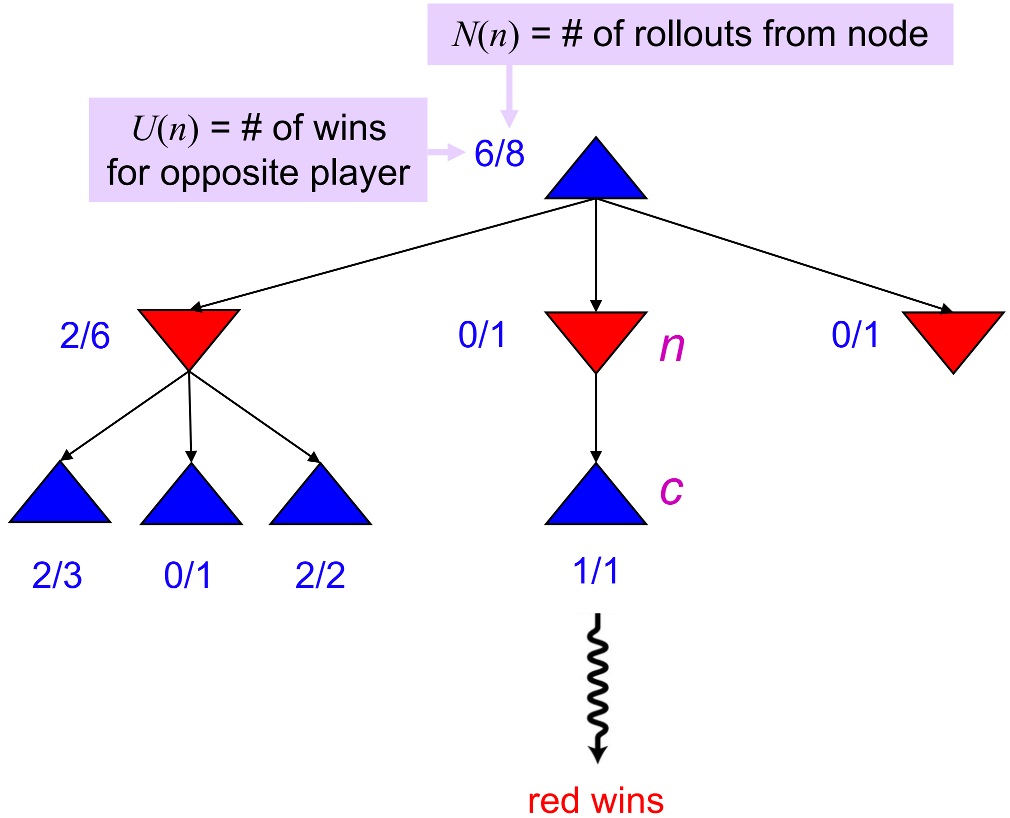
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion**: add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



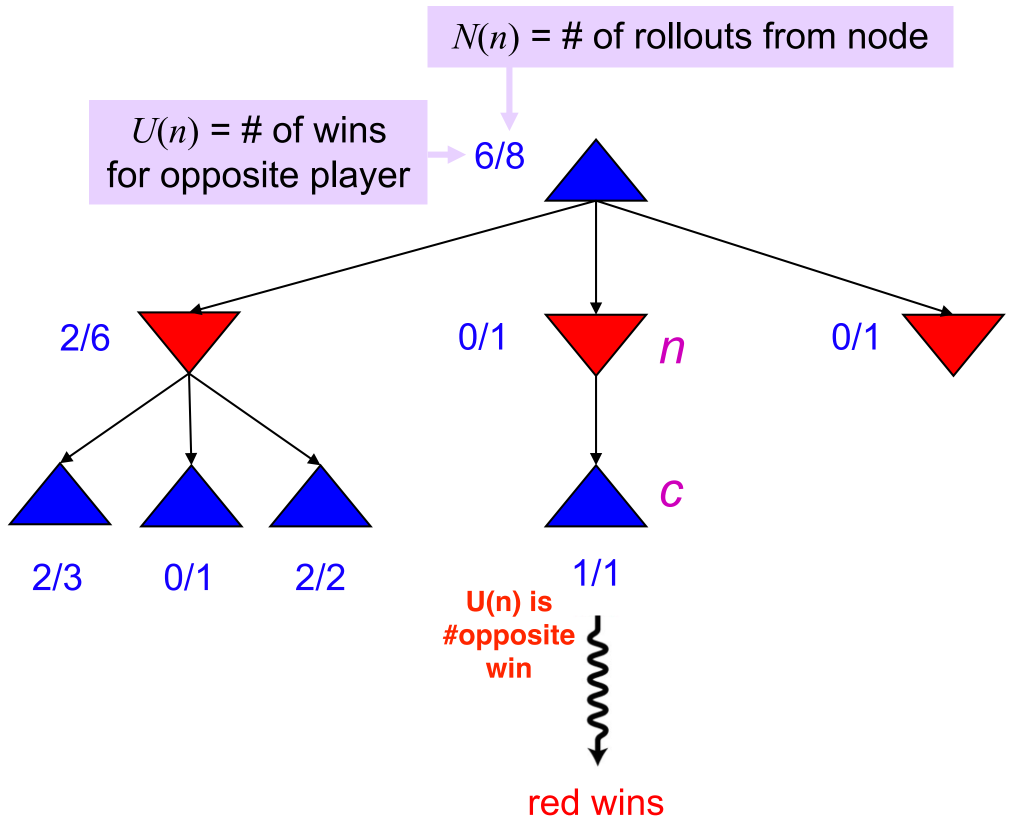
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation**: run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



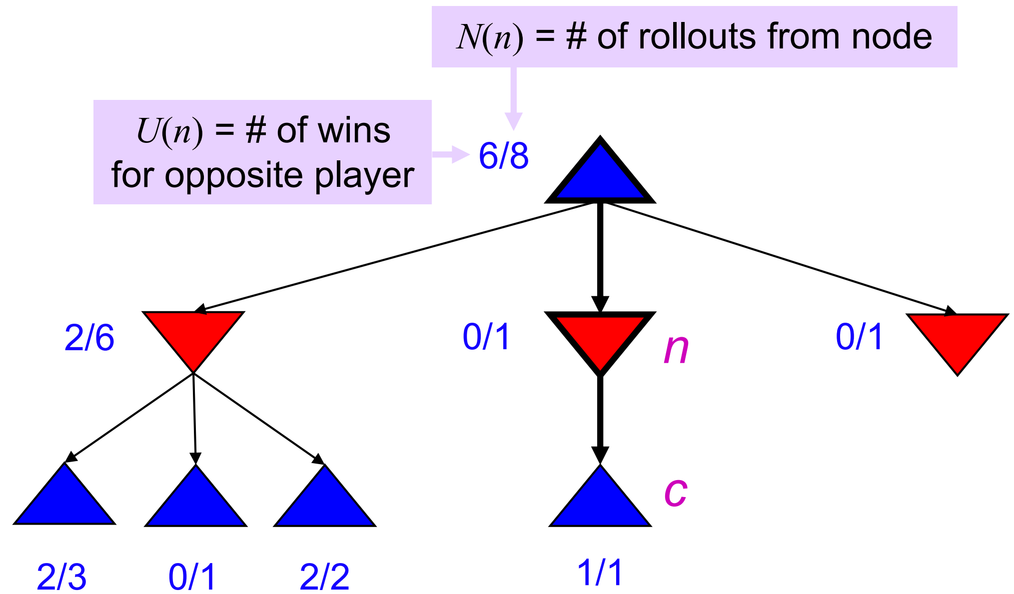
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation**: run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



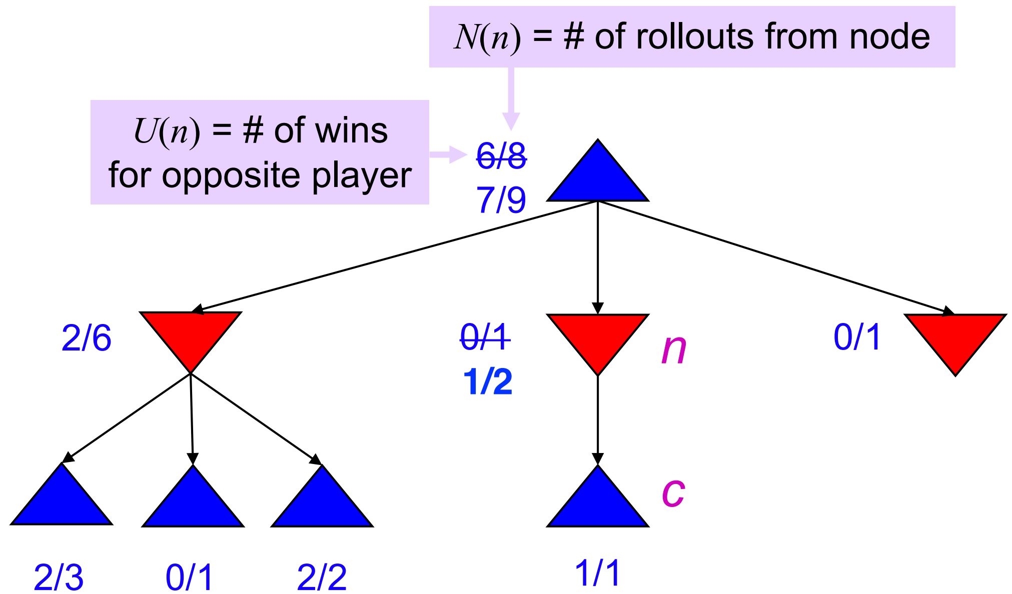
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation**: update U and N counts from *c* back up to the root



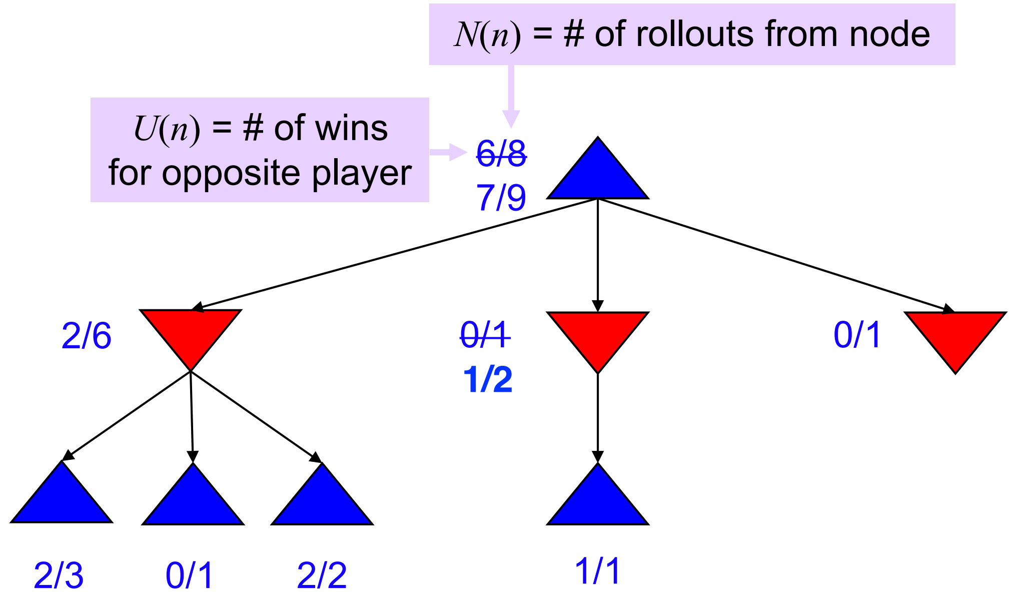
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root



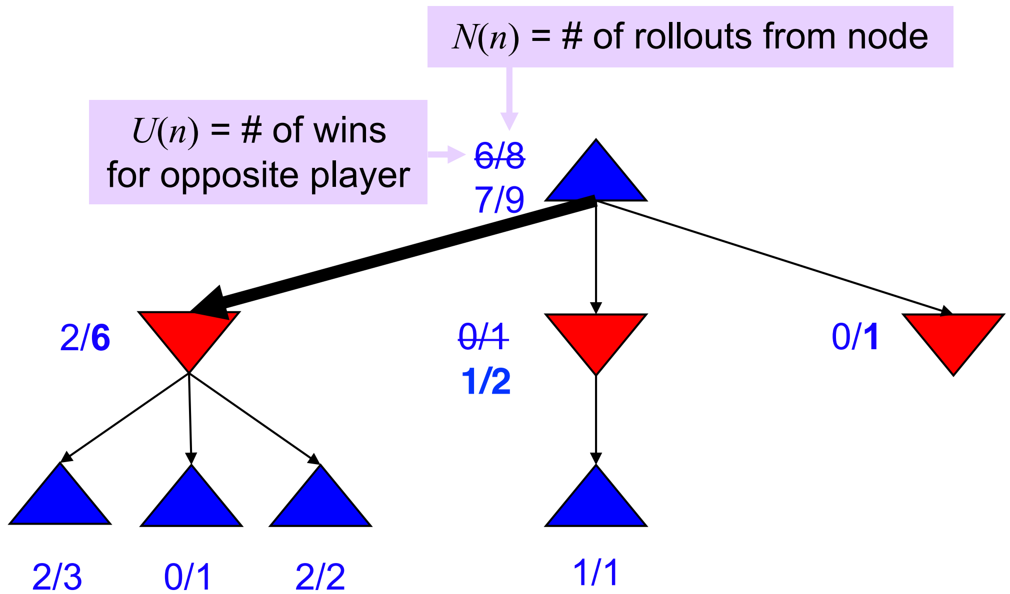
MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root
- Choose the action leading to the child with highest *N*



MCTS Algorithm

- Repeat until out of time:
  - **Selection:** recursively apply UCB to choose a path down to a leaf node *n*
  - **Expansion:** add a new child *c* to *n*
  - **Simulation:** run a rollout from *c*
  - **Backpropagation:** update U and N counts from *c* back up to the root
- Choose the action leading to the child with highest *N*



MCTS Summary

- MCTS is currently the most common tool for solving hard search problems
- Why?
  - Time complexity independent of b and m
  - No need to design evaluation functions (general-purpose & easy to use)
- Solution quality depends on number of rollouts N
  - Theorem: as $N \to \infty$ UCT selects the minimax move
- Example of using random sampling in an algorithm
  - Broadly called *Monte Carlo* methods
- MCTS can be improved further with machine learning