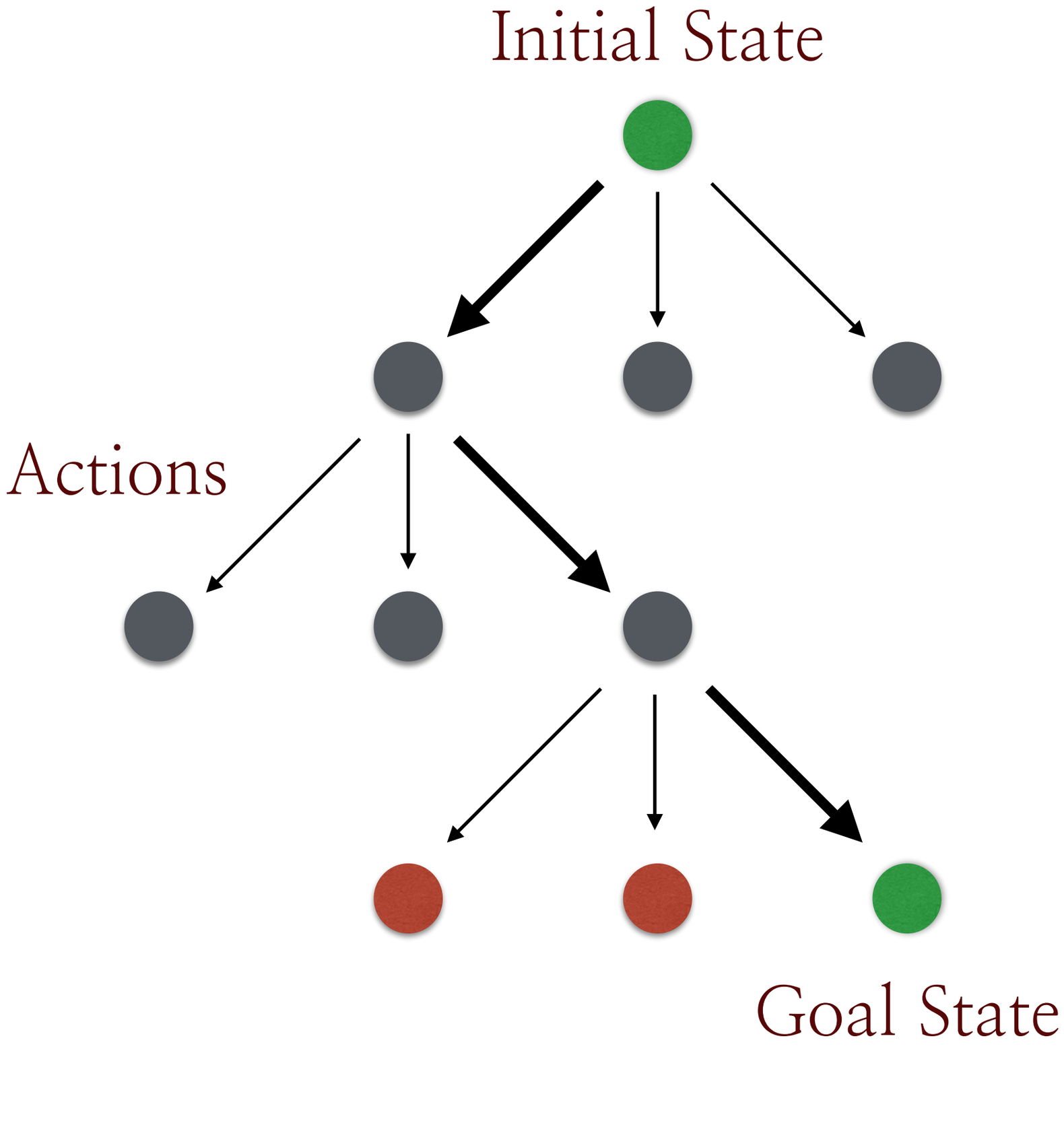
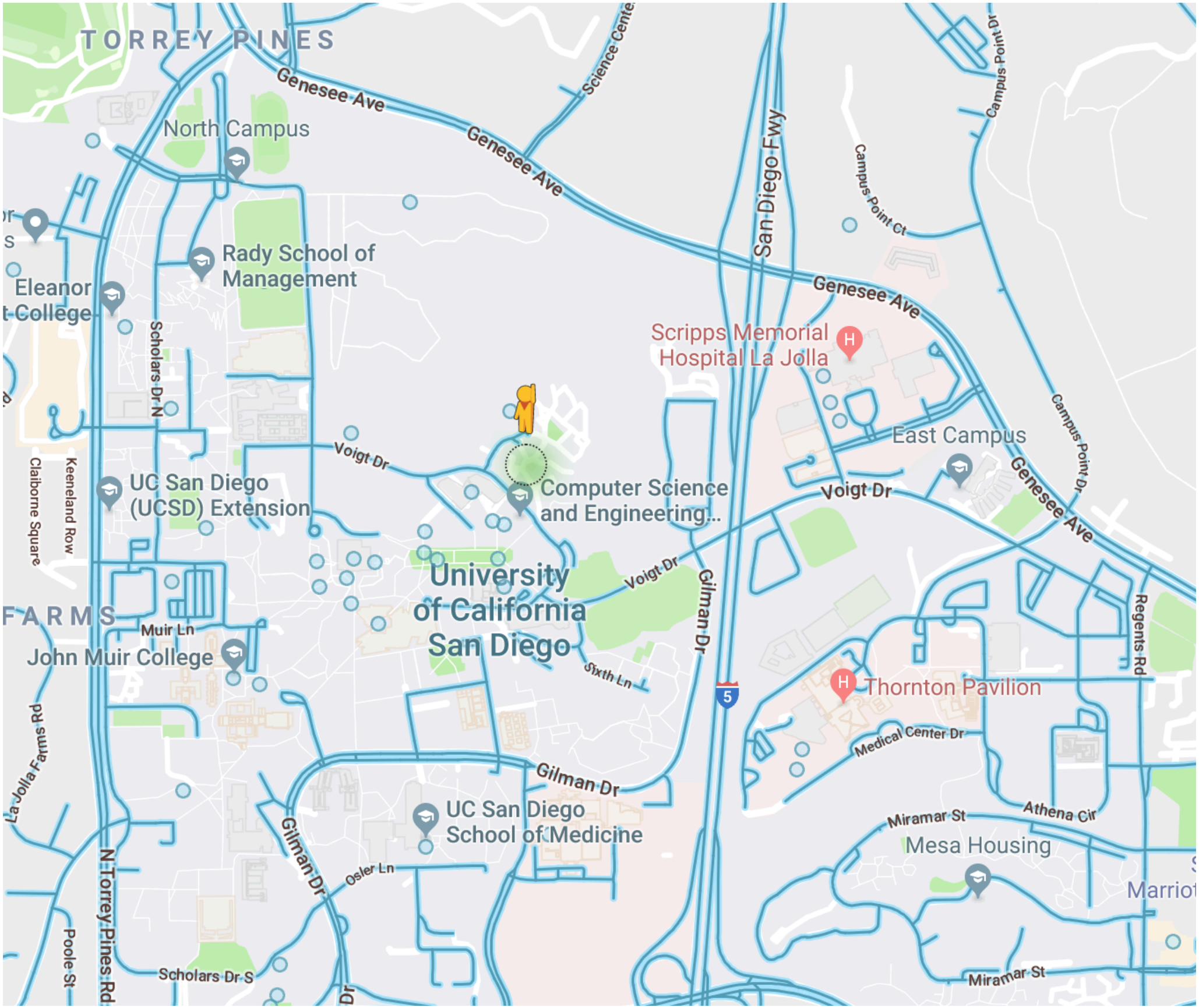
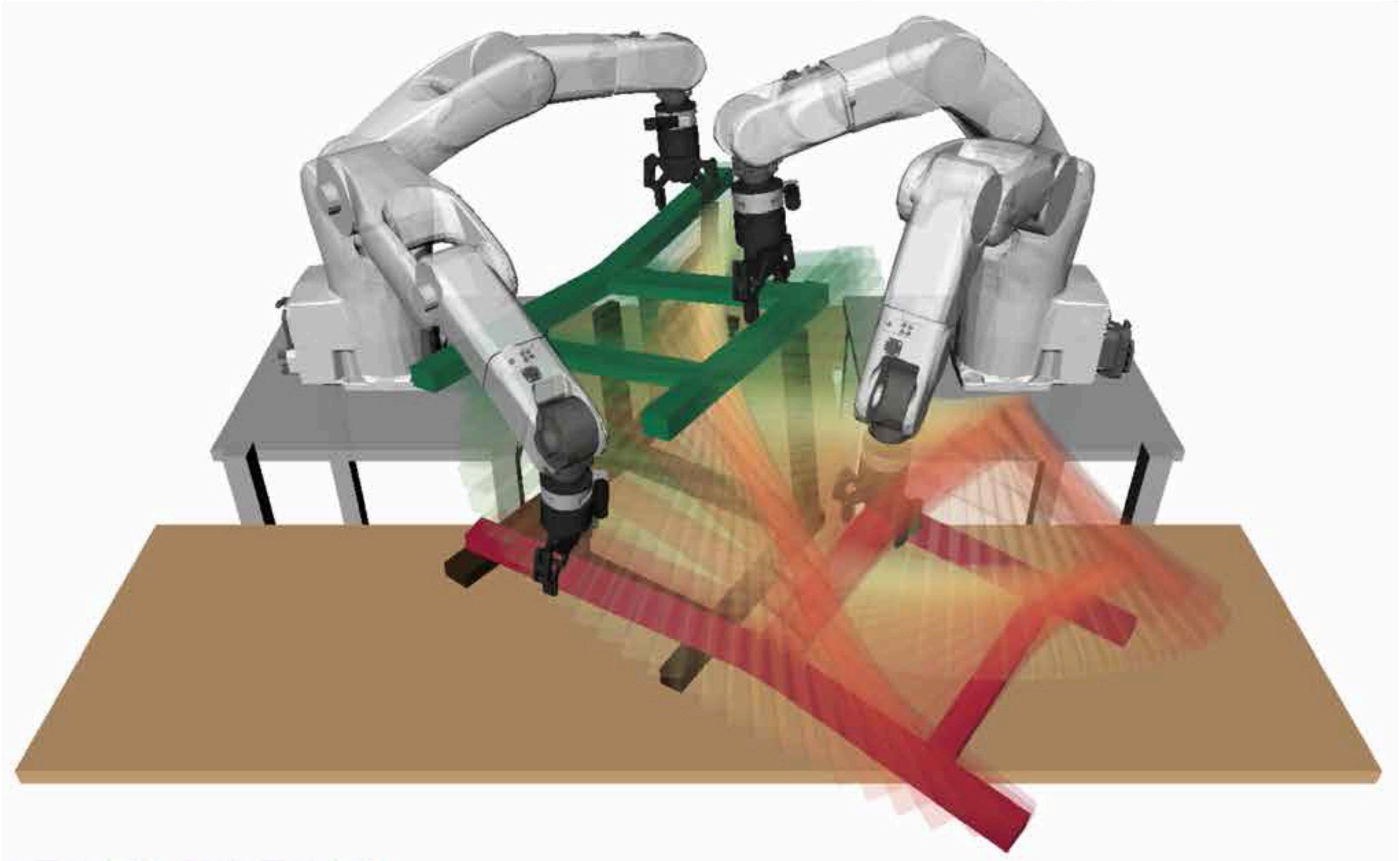
Classical Search

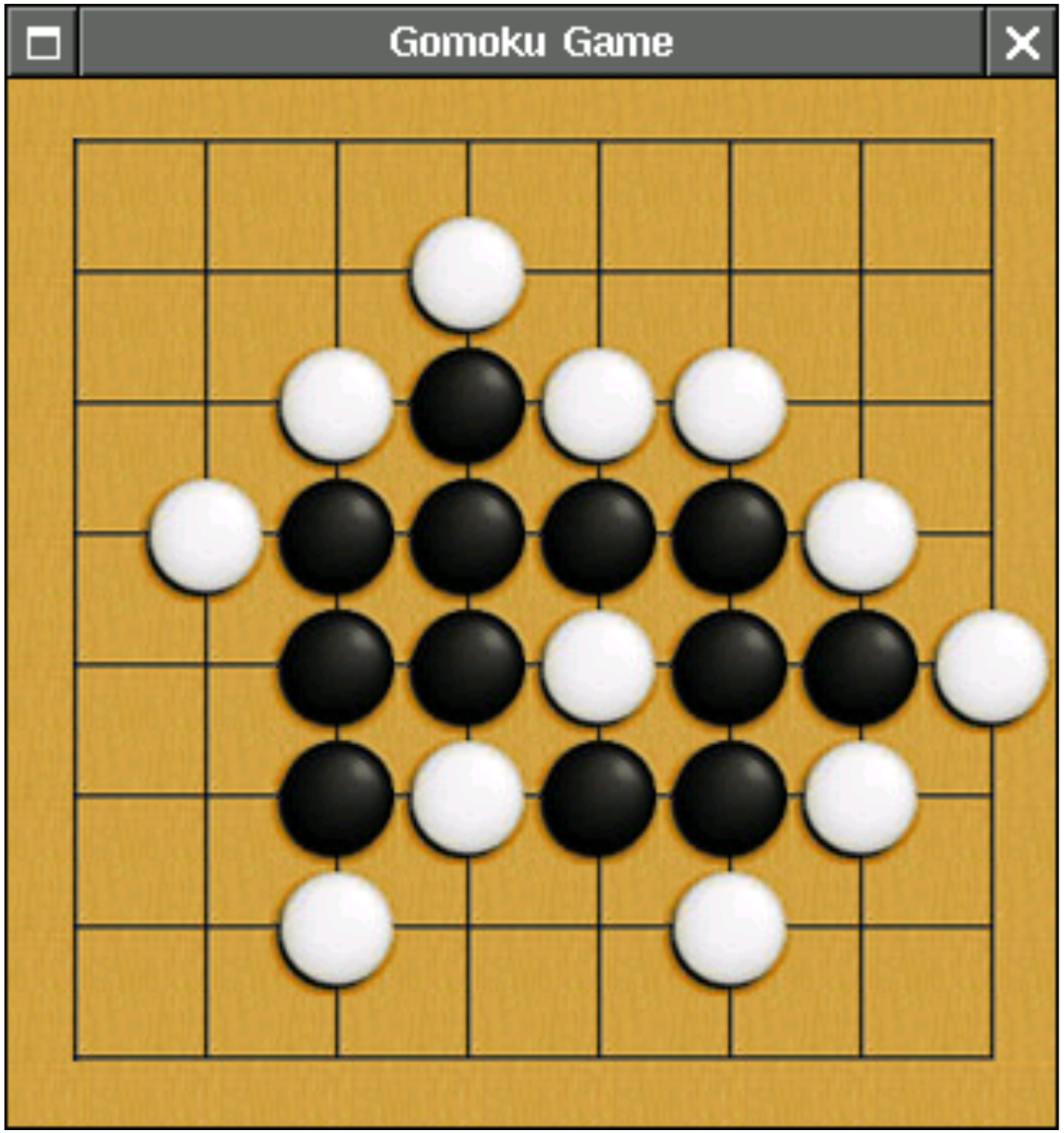
UCSD CSE 257

Sicun Gao

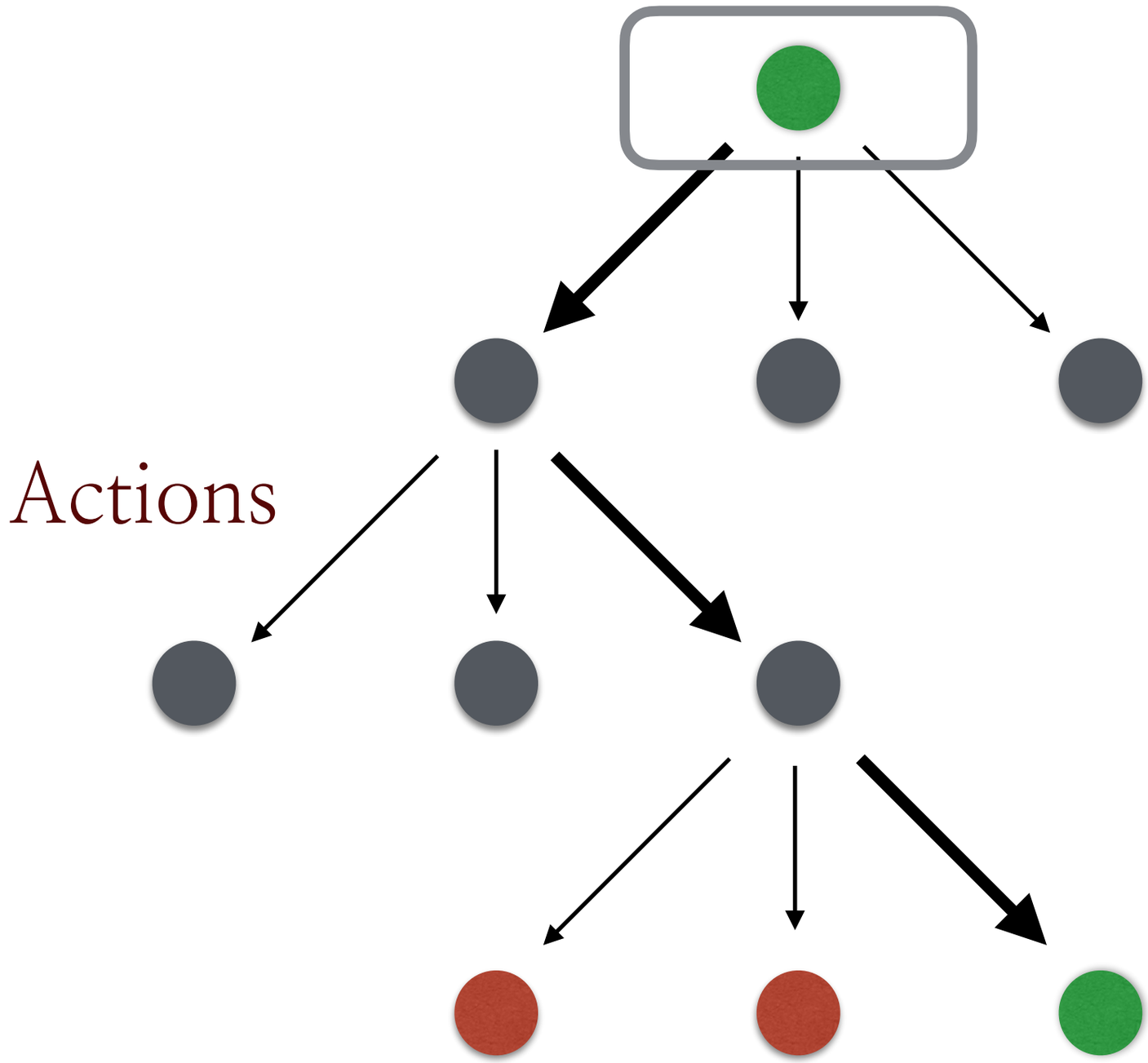

Initial State
Actions
Goal State


TORREY PINES
Genesee Ave
Science Cente
North Campus
Genesee Ave
San Diego Fwy
Campus Point Dr
Campus Point Ct
Rady School of Management
Eleanor
t College
Scholars Dr N
Genesee Ave
Scripps Memorial Hospital La Jolla
East Campus
Campus Point Dr
Voigt Dr
Claiborne Square
Keeneland Row
UC San Diego (UCSD) Extension
Computer Science and Engineering...
Voigt Dr
Genesee Ave
University of California San Diego
Voigt Dr
Gilman Dr
Regents Rd
FARMS
Muir Ln
John Muir College
Sixth Ln
5
Thornton Pavilion
Medical Center Dr
La Jolla Farms Rd
Gilman Dr
UC San Diego School of Medicine
Miramar St
Athena Cir
Mesa Housing
N Torrey Pines Rd
Gilman Dr
Osler Ln
Poole St
Scholars Dr S
Miramar St




Gomoku Game

Initial State



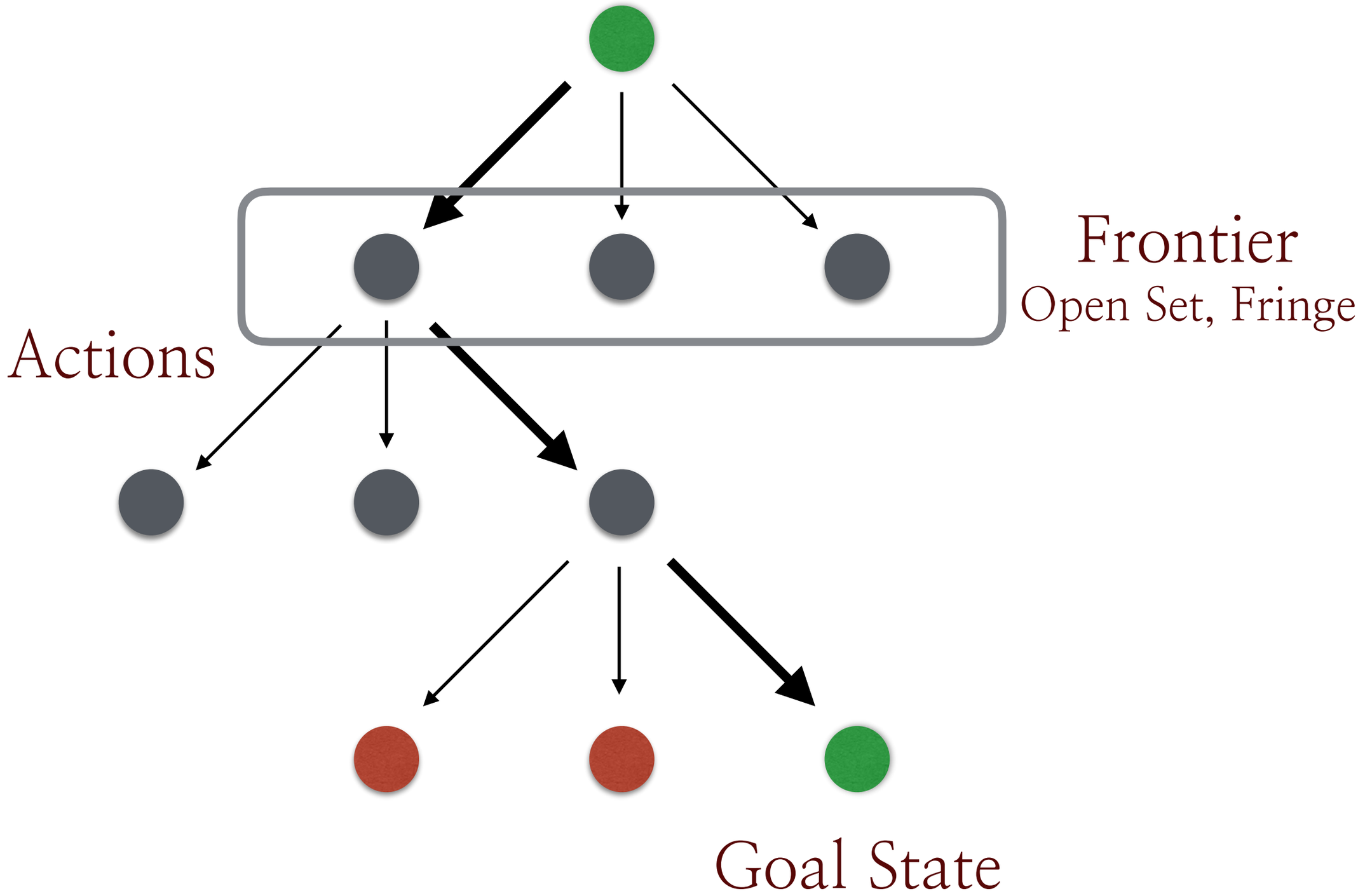
Frontier

Open Set, Fringe

Actions

Goal State

Initial State



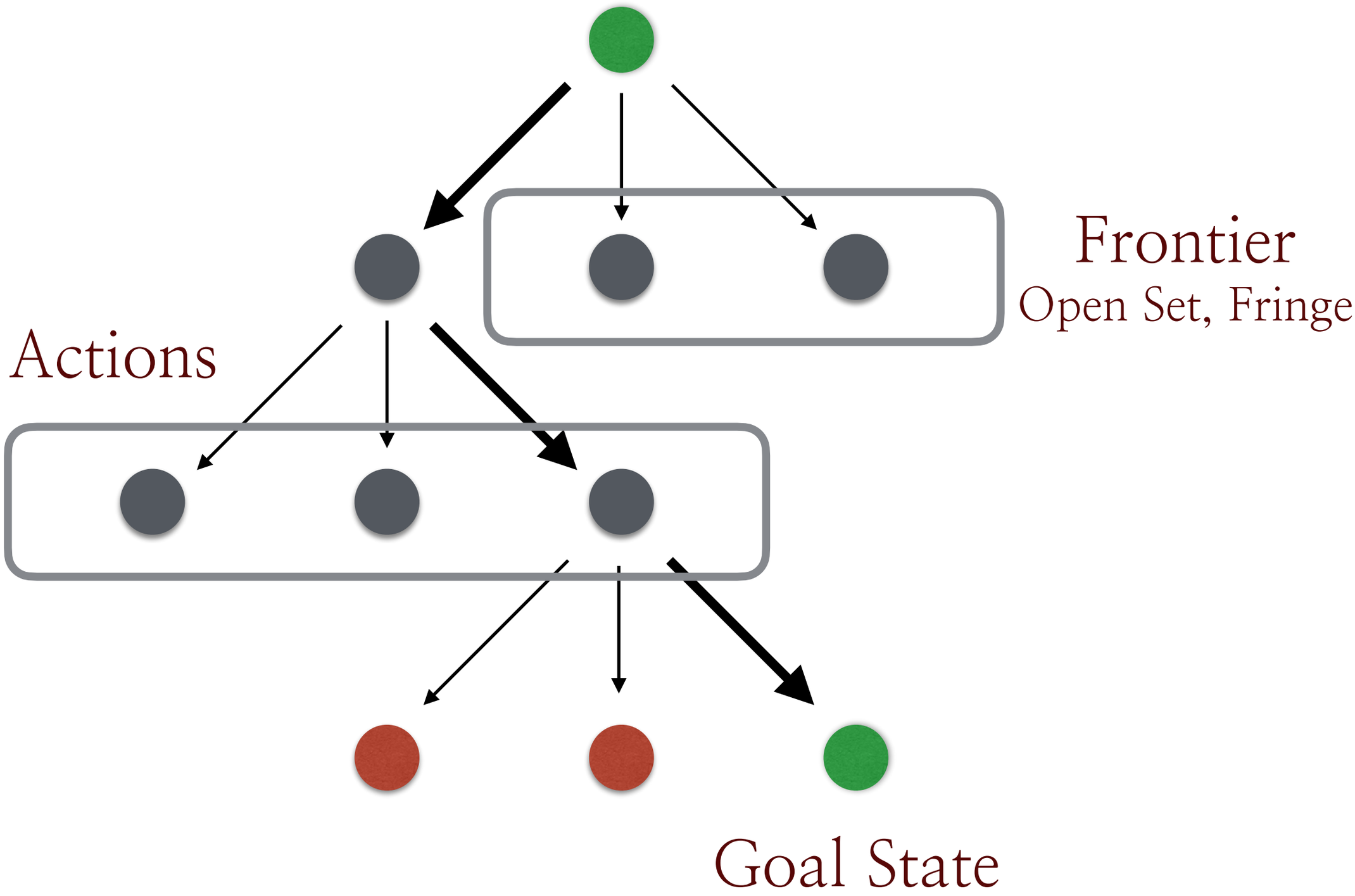
Frontier

Open Set, Fringe

Actions

Goal State

Initial State



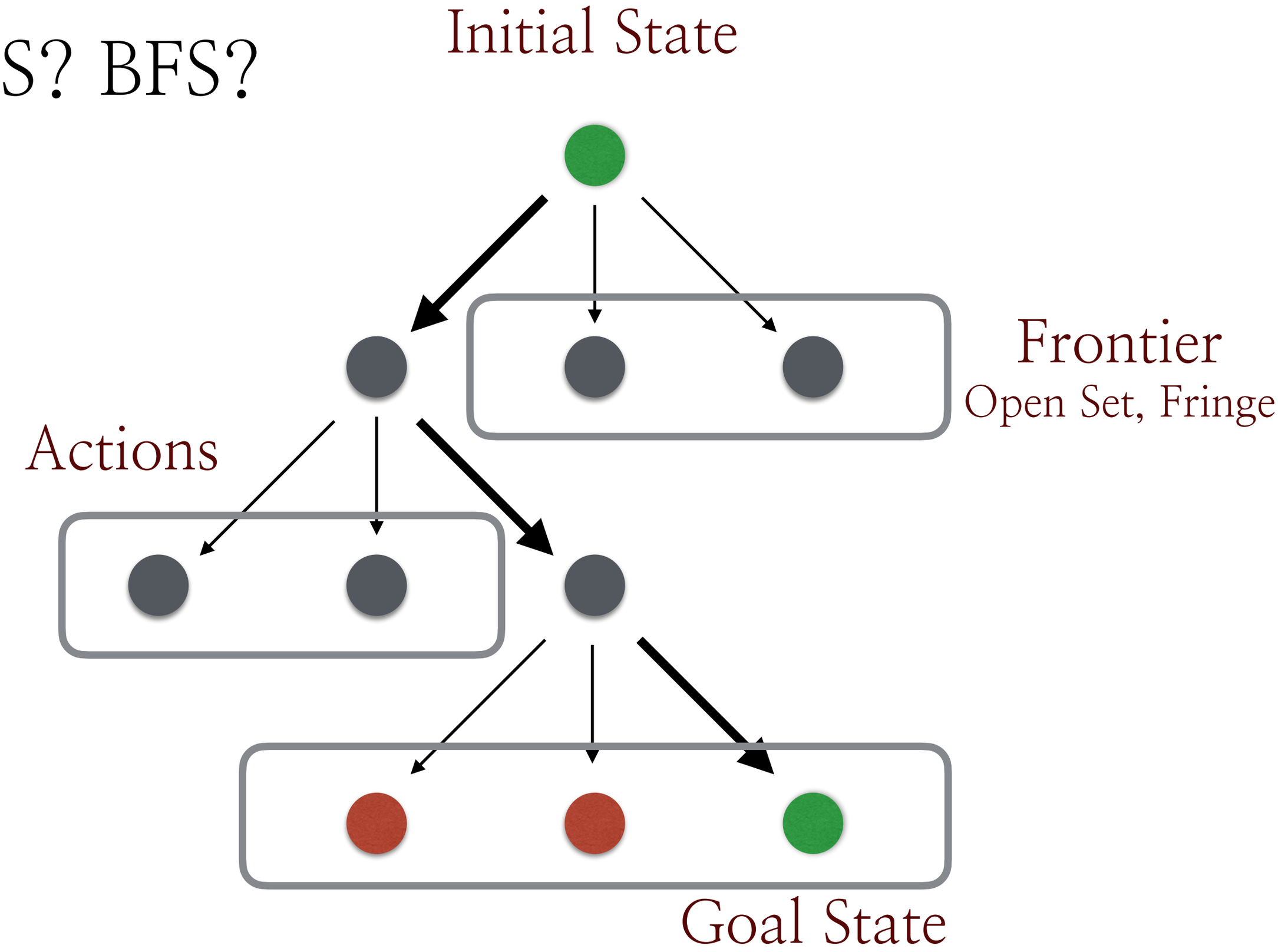
Frontier

Open Set, Fringe

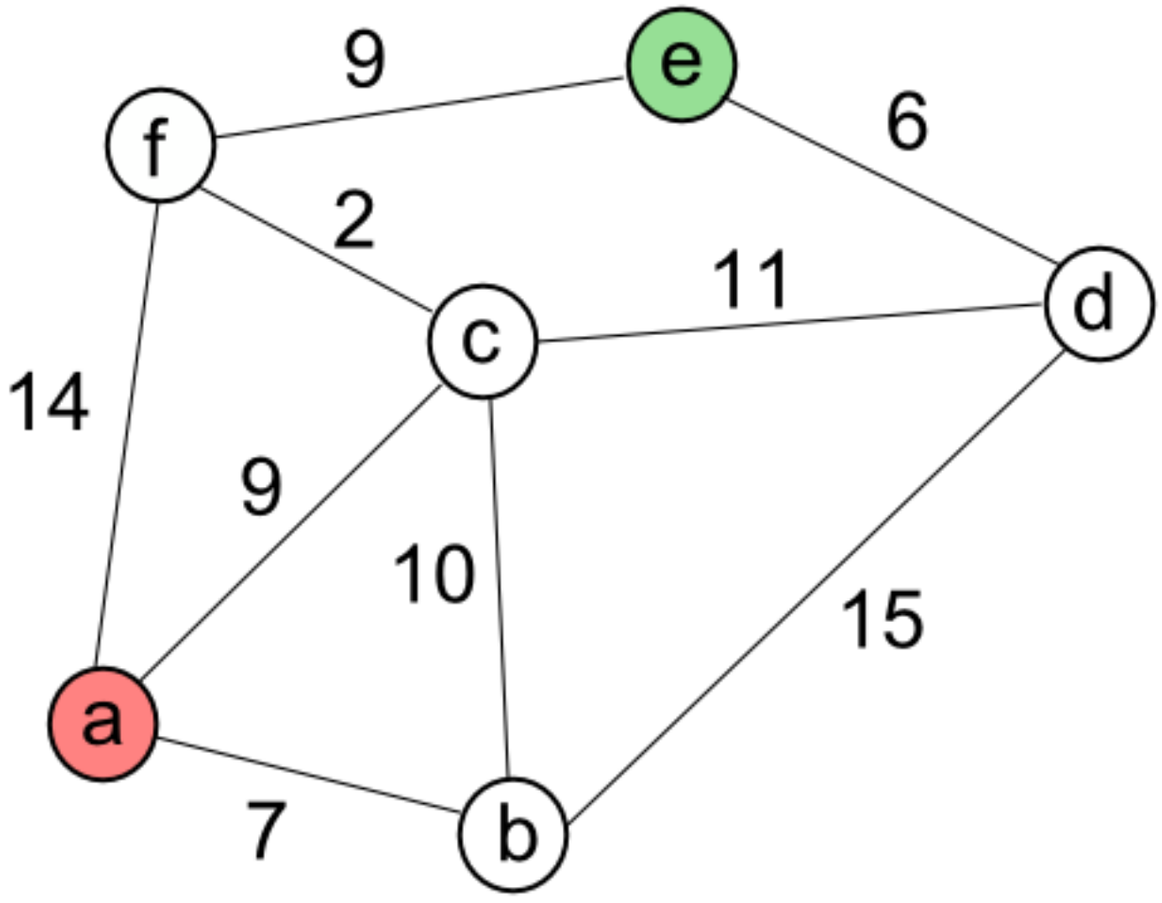
Actions

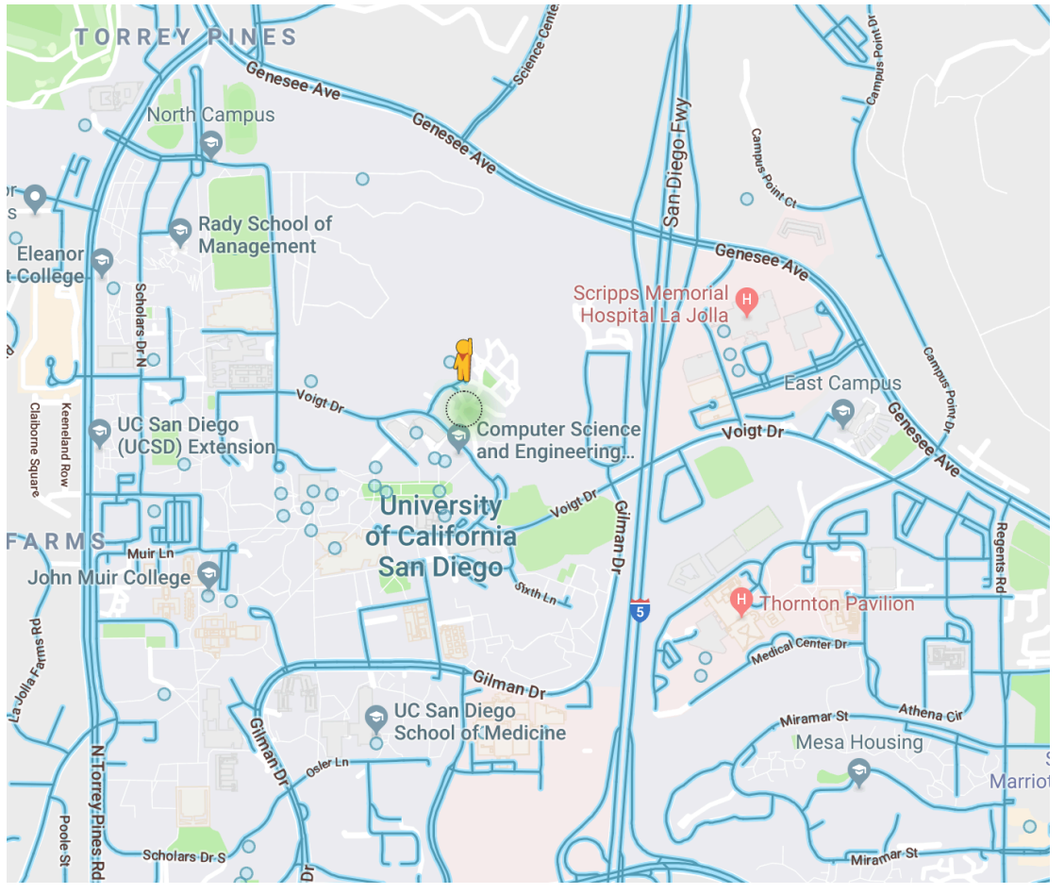
Goal State

DFS? BFS?

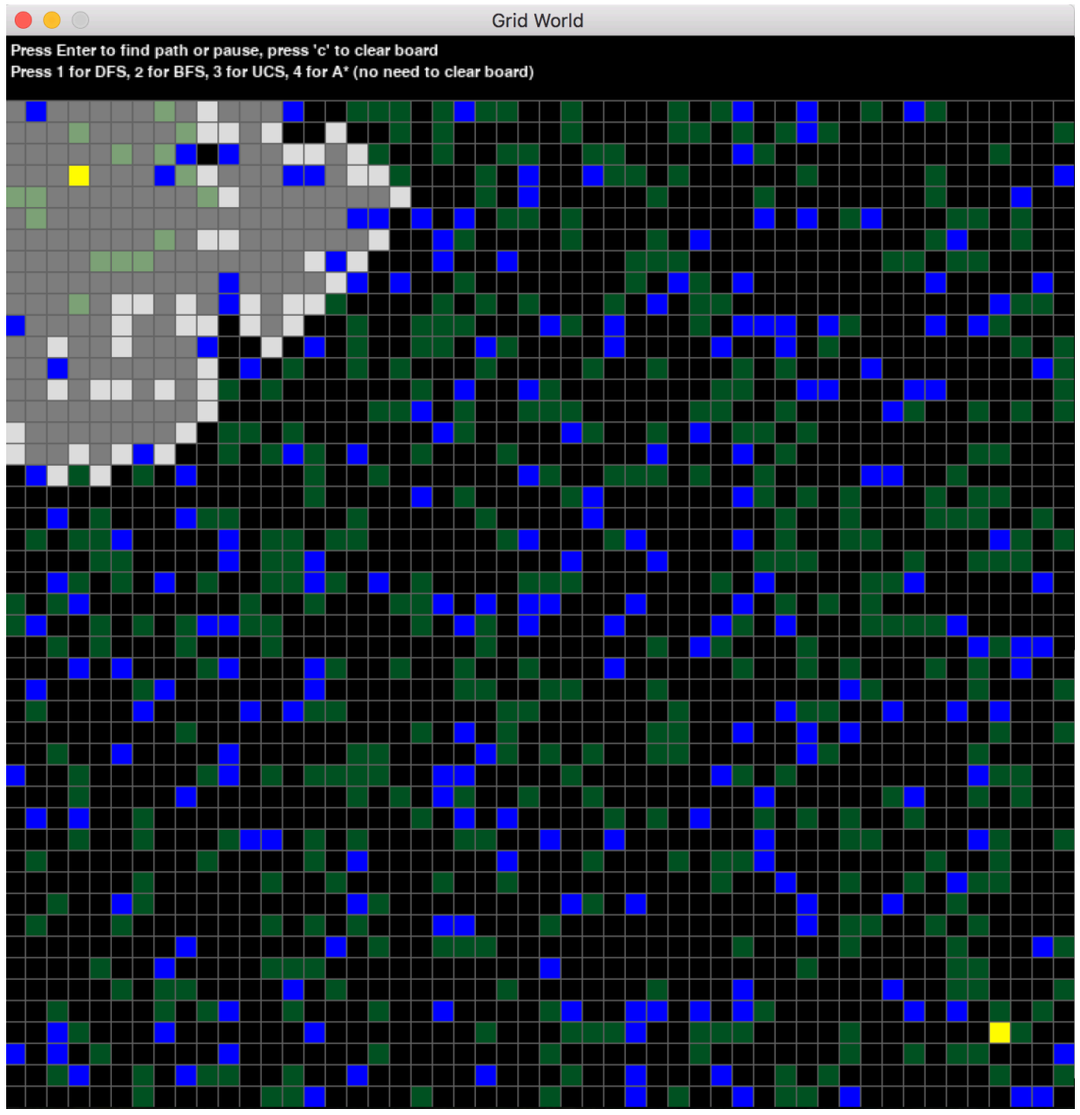


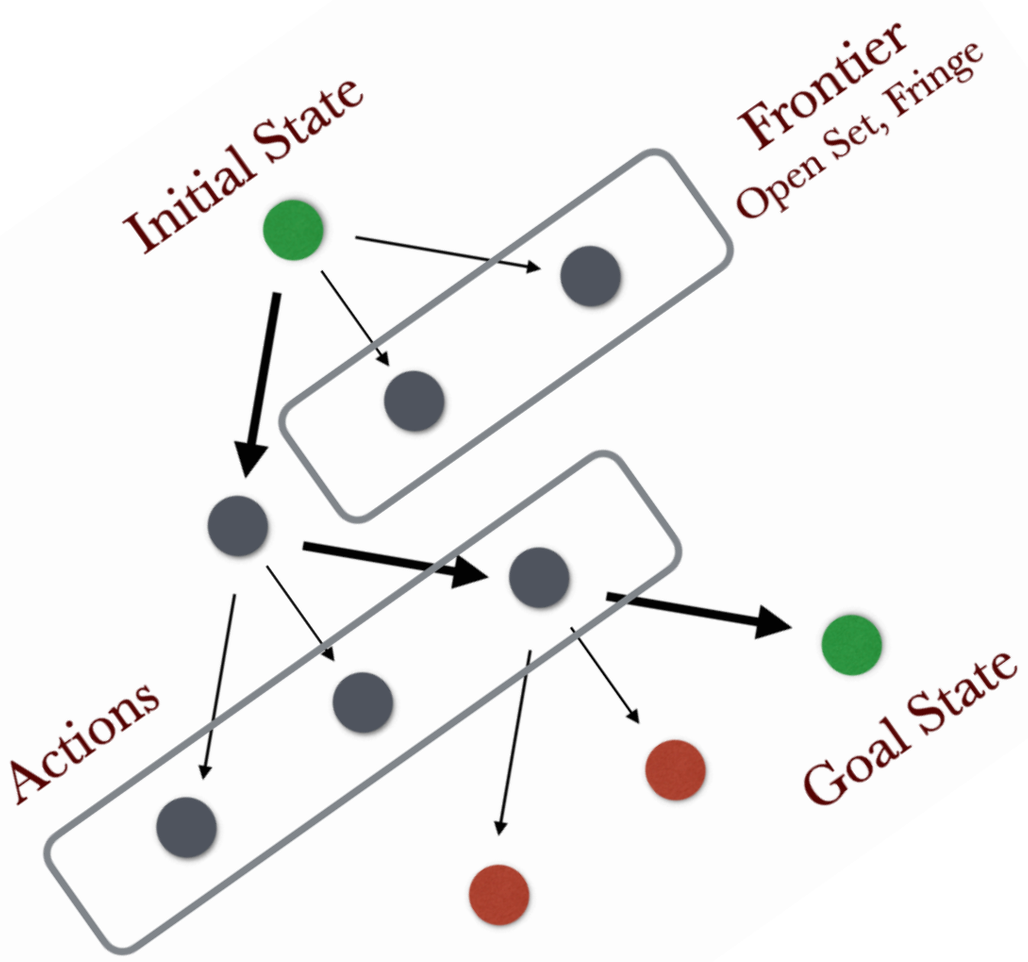
Search in Graphs





Search in Graphs

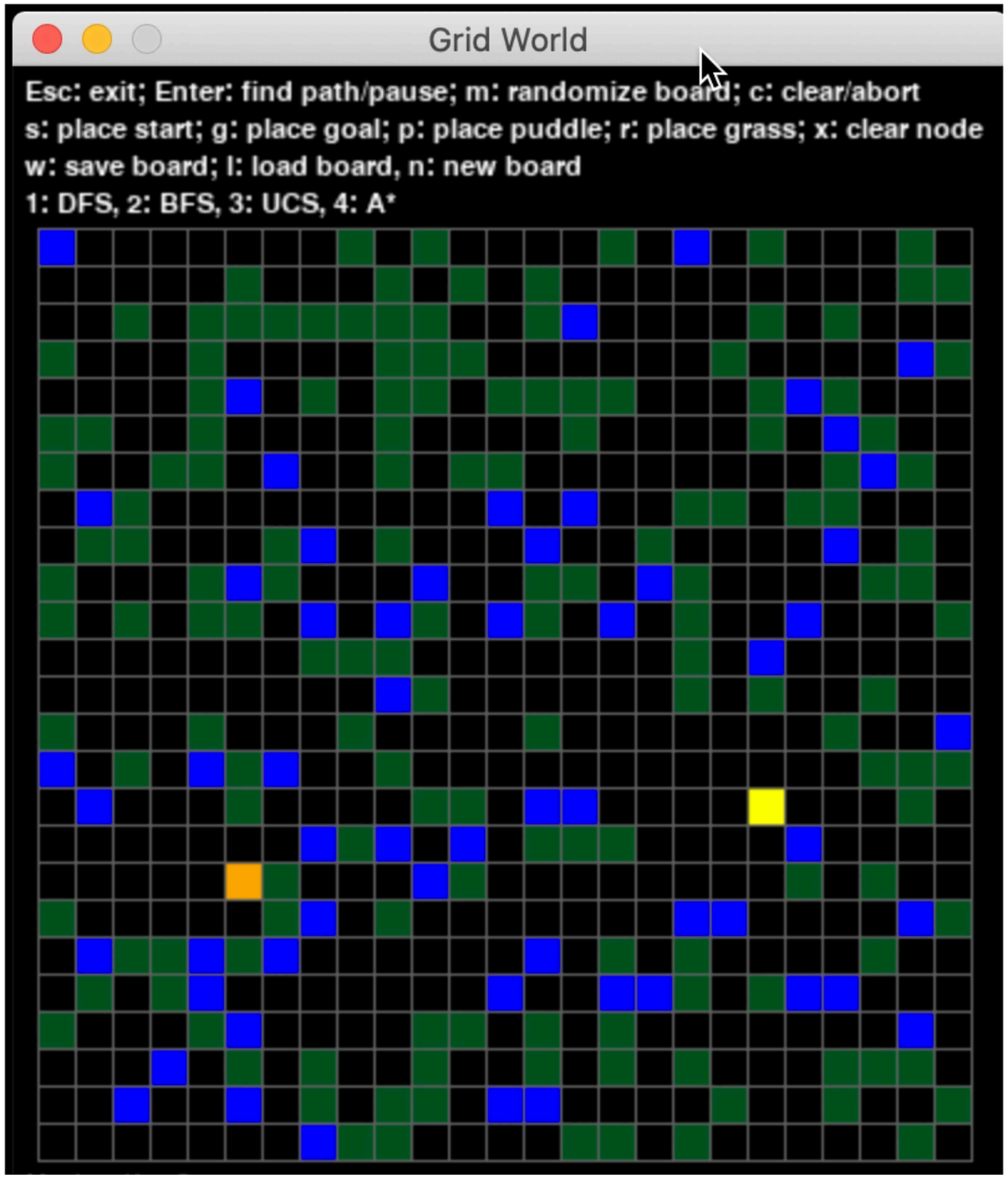




Search in Graphs

Danger: turning finite graphs into infinite trees

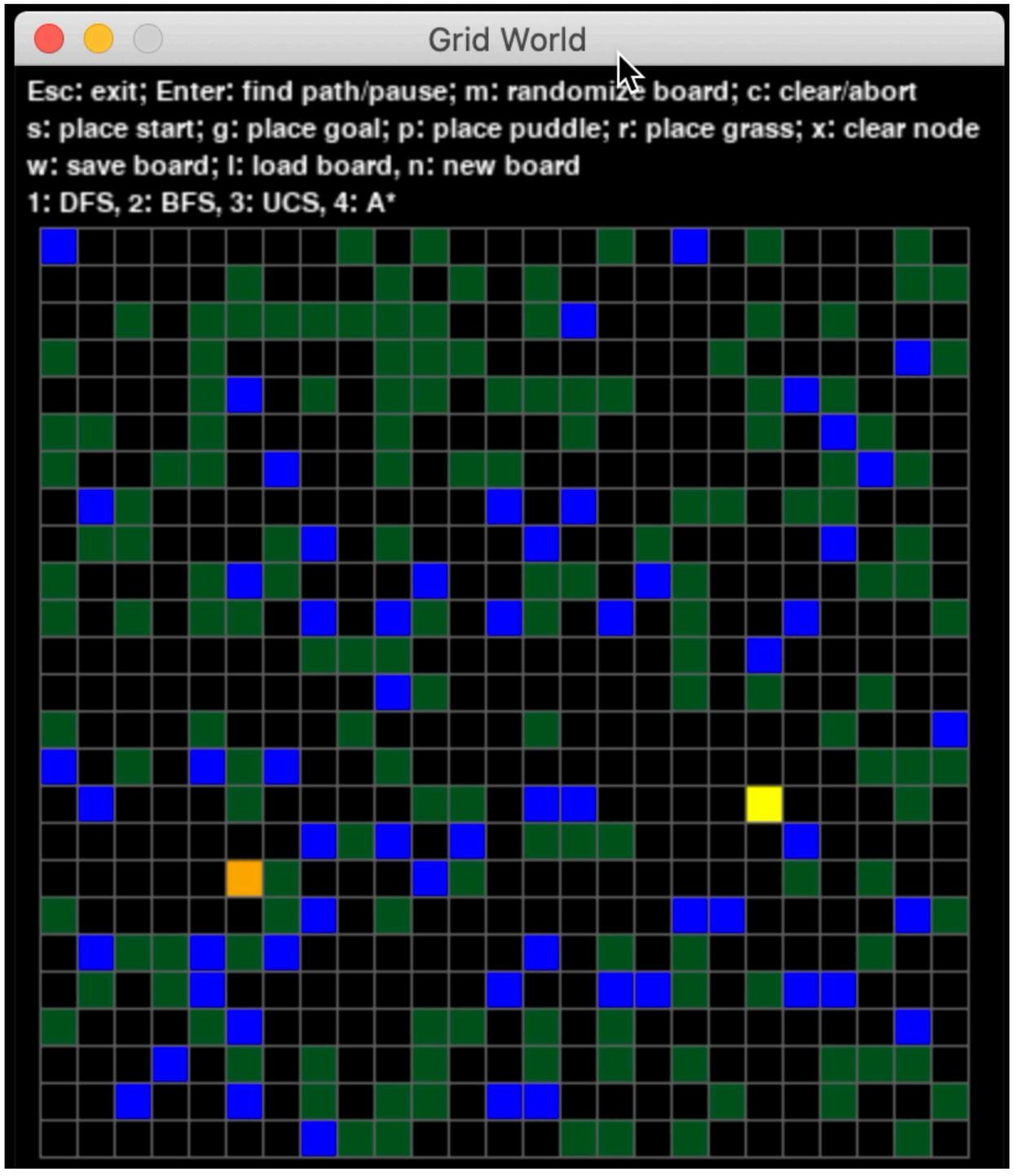
Make sure that you don't revisit nodes that are already explored



DFS on graphs (Complete? Optimal?)

```
frontier = empty_stack.push(start)
explored = empty_stack
while frontier:
    current = frontier.pop()
    explored.push(current)
    for node in current.children():
        if node == goal:
            return path '''need bookkeeping'''
        if node in explored or node in frontier:
            continue
        else:
            frontier.push(node)
return failed
```

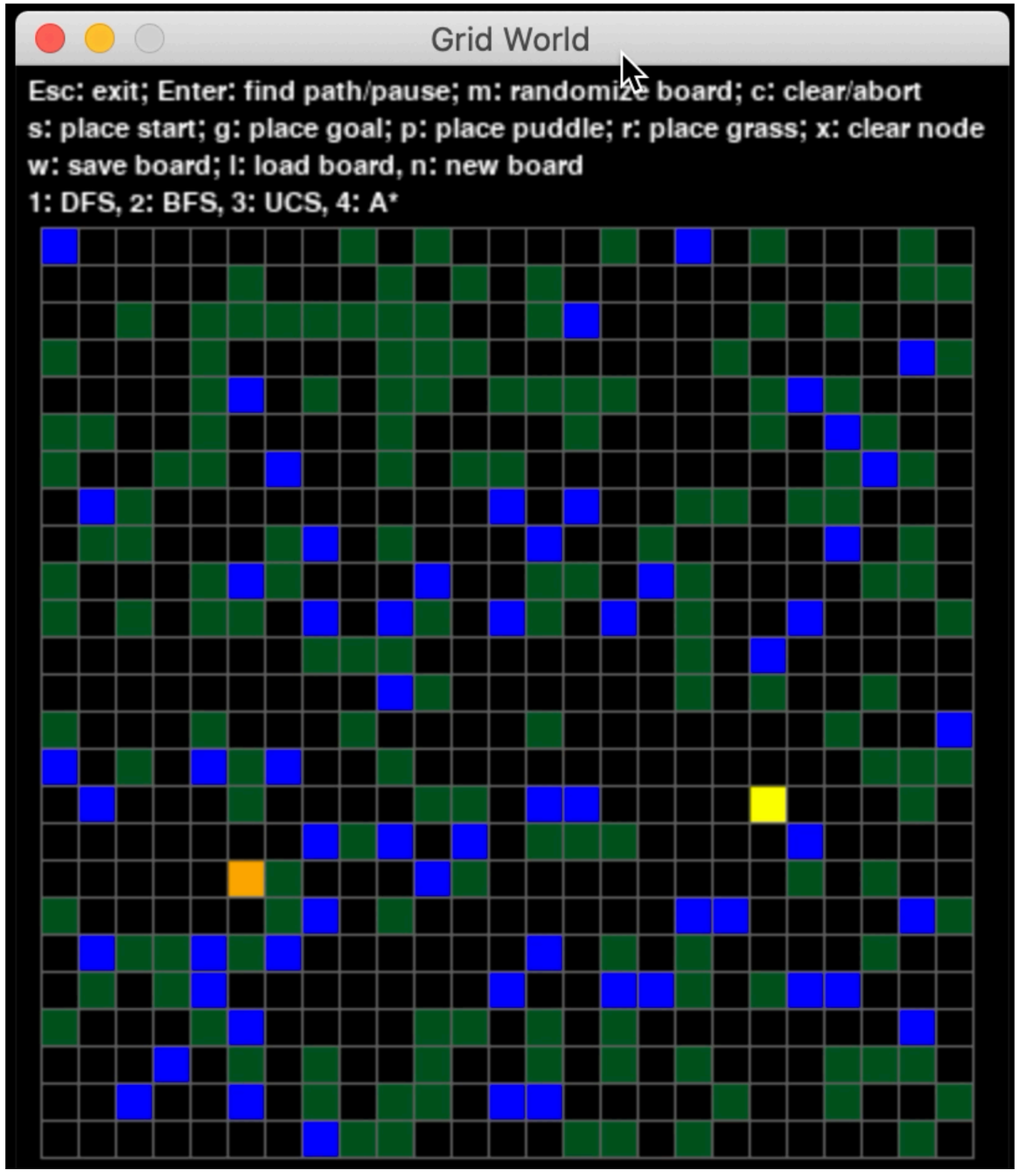
DFS on graphs (Complete? Optimal?)



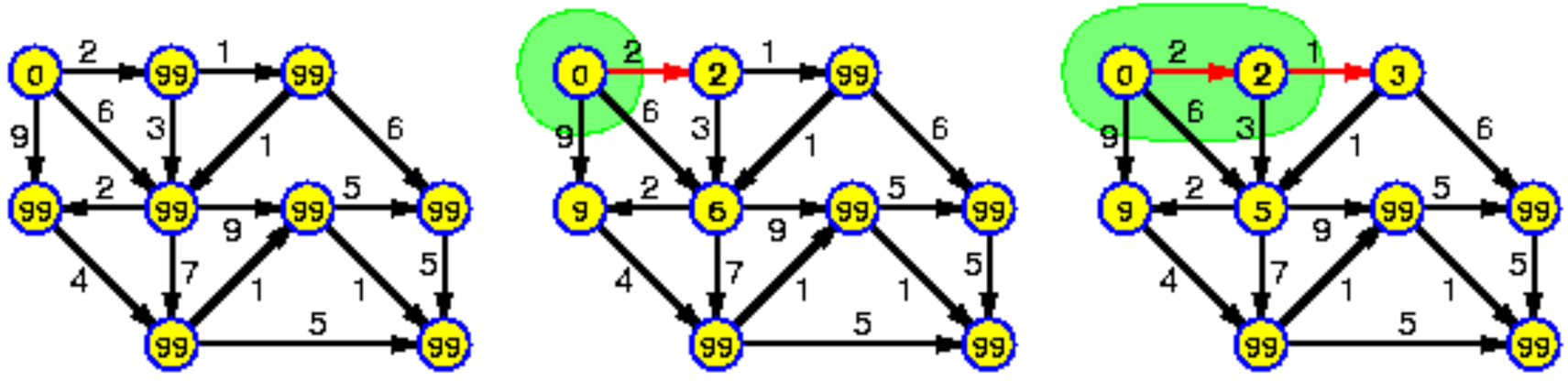
BFS on graphs (Complete? Optimal?)

```
frontier = empty_queque.enqueue(start)
explored = empty_stack
while frontier:
    current = frontier.dequeue()
    explored.push(current)
    for node in current.children():
        if node == goal:
            return path '''need bookkeeping'''
        if node in explored or node in frontier:
            continue
        else:
            frontier.enqueue(node)
return failed
```

BFS on graphs (Complete? Optimal?)



Dijkstra/Uniform Cost Search

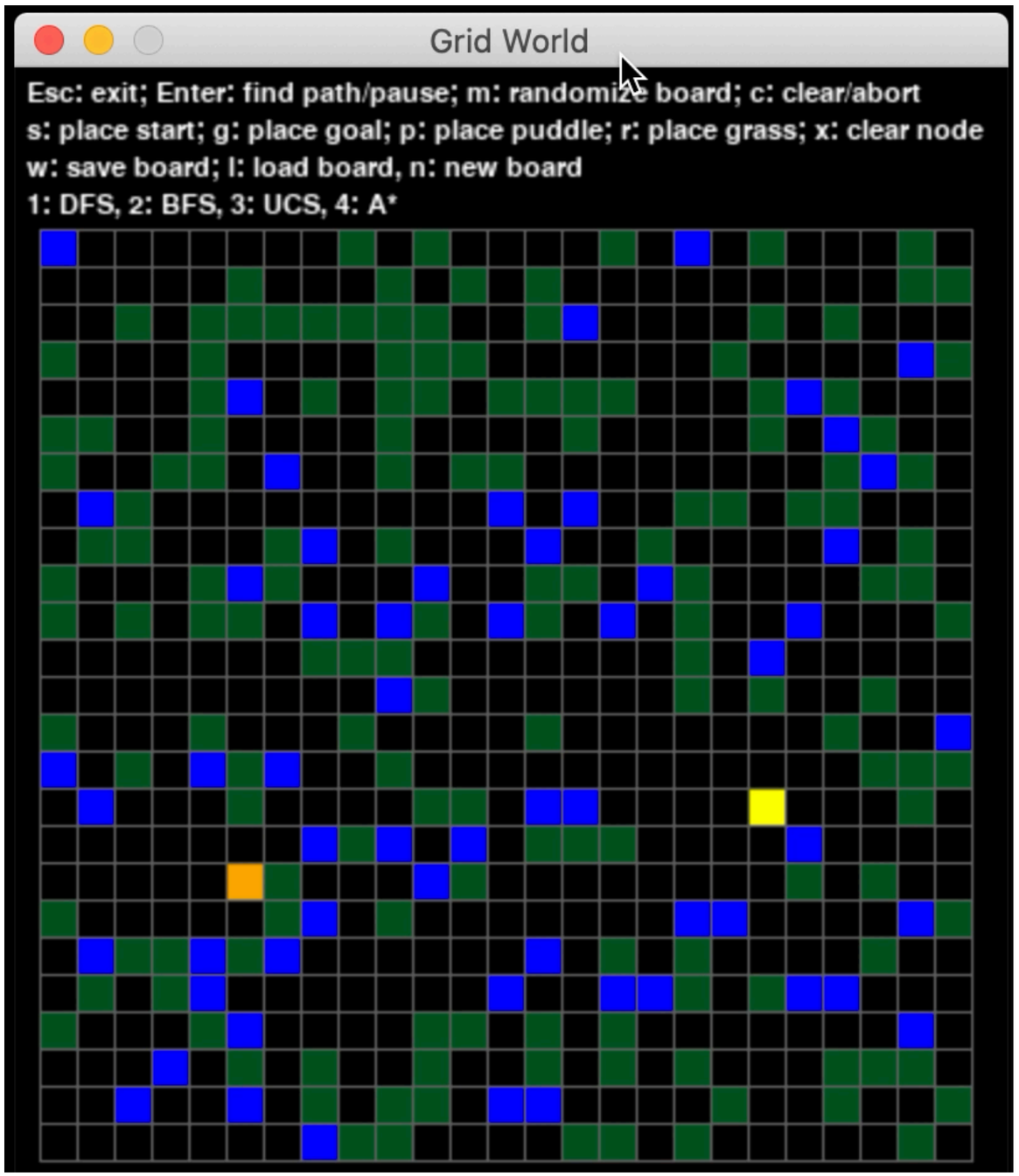


Dijkstra/Uniform Cost Search

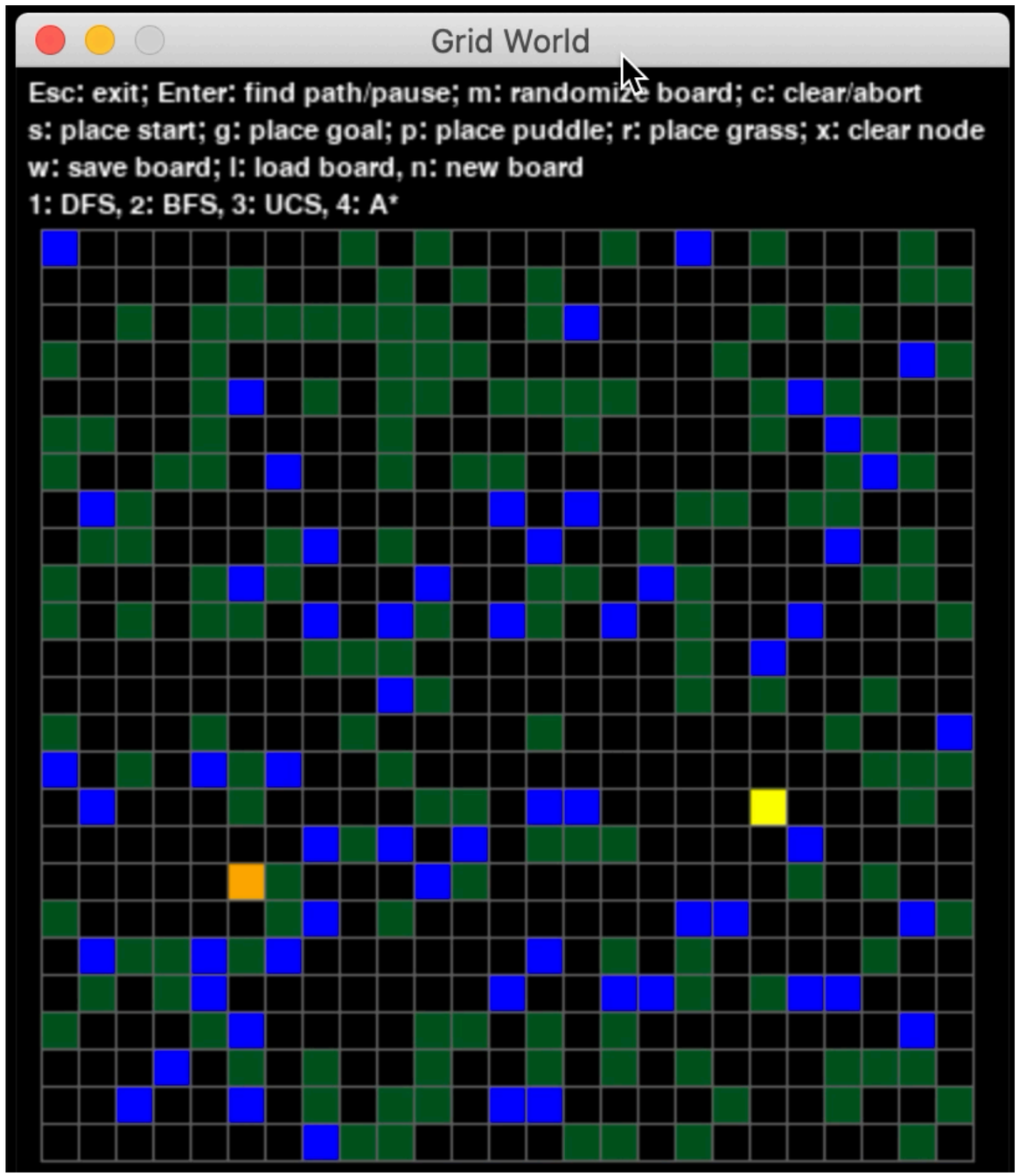
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with *node* as the only element
explored ← an empty set
loop do
  if EMPTY?(*frontier*) **then return** failure
  node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  add *node*.STATE to *explored*
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    child ← CHILD-NODE(*problem*, *node*, *action*)
    if *child*.STATE is not in *explored* or *frontier* **then**
      frontier ← INSERT(*child*, *frontier*)
    else if *child*.STATE is in *frontier* with higher PATH-COST **then**
      replace that *frontier* node with *child*

pop a frontier node, maintain explored
push children, maintain frontier


Grid World
Esc: exit; Enter: find path/pause; m: randomize board; c: clear/abort
s: place start; g: place goal; p: place puddle; r: place grass; x: clear node
w: save board; l: load board, n: new board
1: DFS, 2: BFS, 3: UCS, 4: A*

BFS


Grid World
Esc: exit; Enter: find path/pause; m: randomize board; c: clear/abort
s: place start; g: place goal; p: place puddle; r: place grass; x: clear node
w: save board; l: load board, n: new board
1: DFS, 2: BFS, 3: UCS, 4: A*

UCS

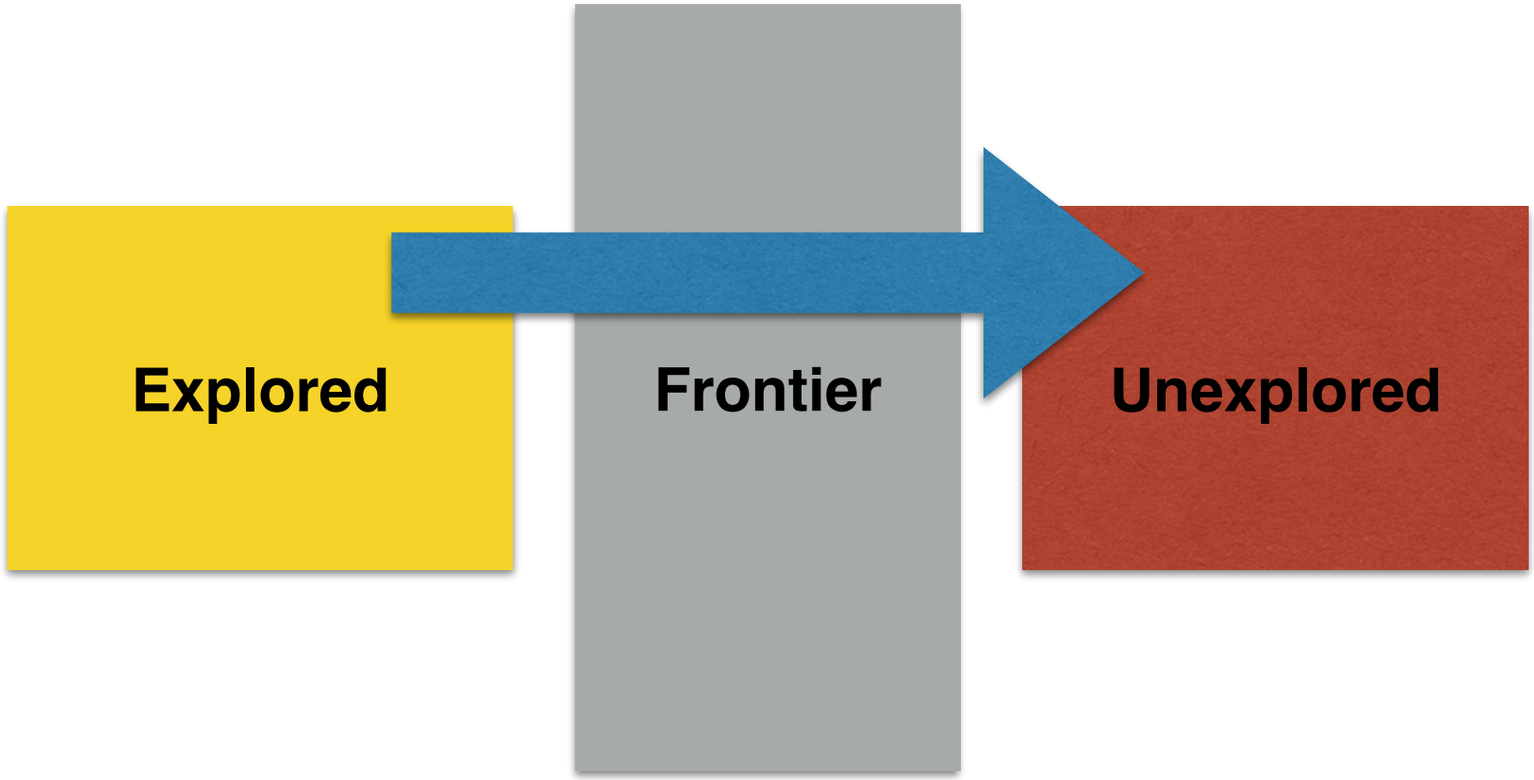
Analysis of UCS: Optimality

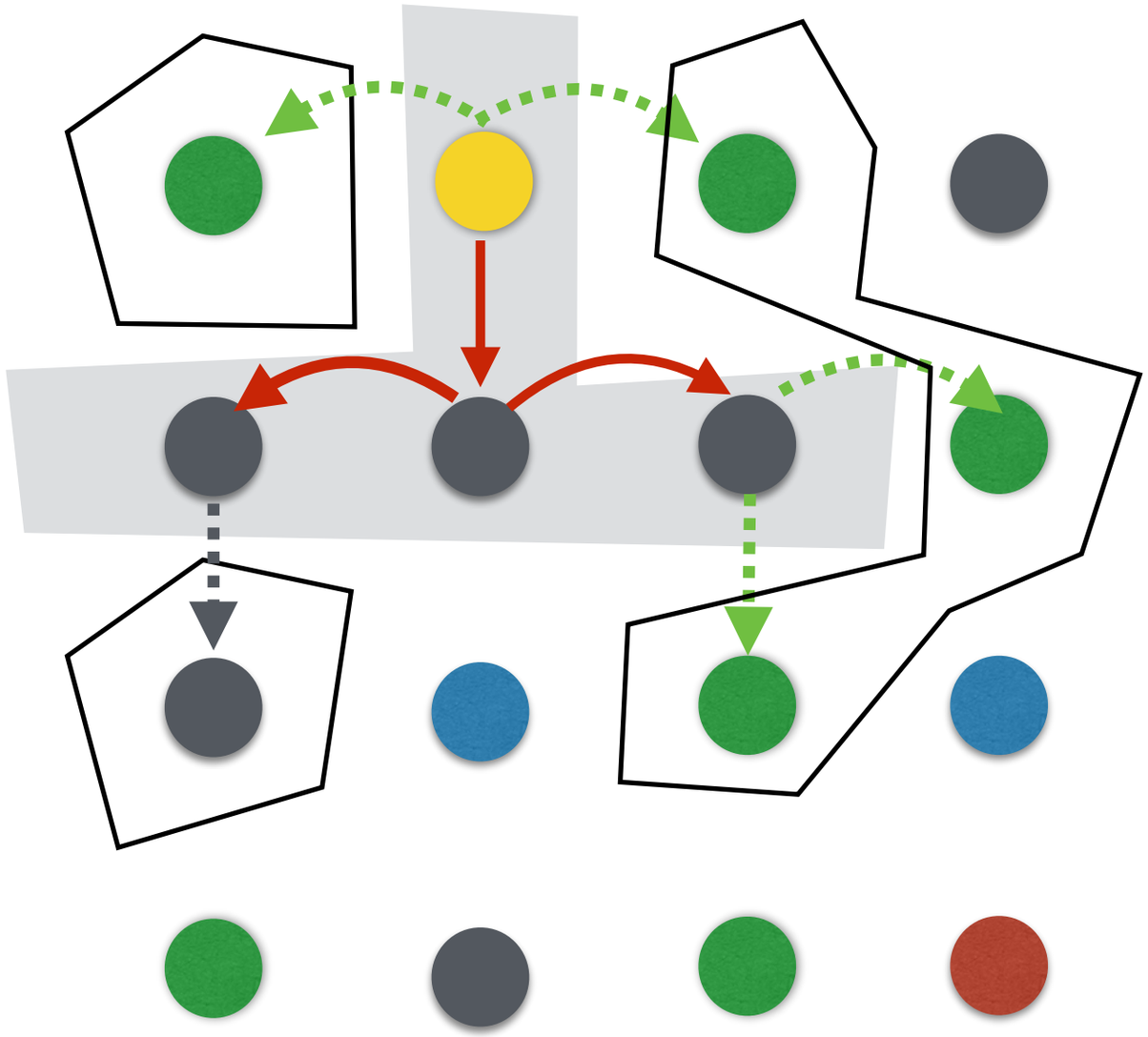
- When any node is expanded, UCS has found the optimal path to it (i.e., with minimum cost).

How to prove this?

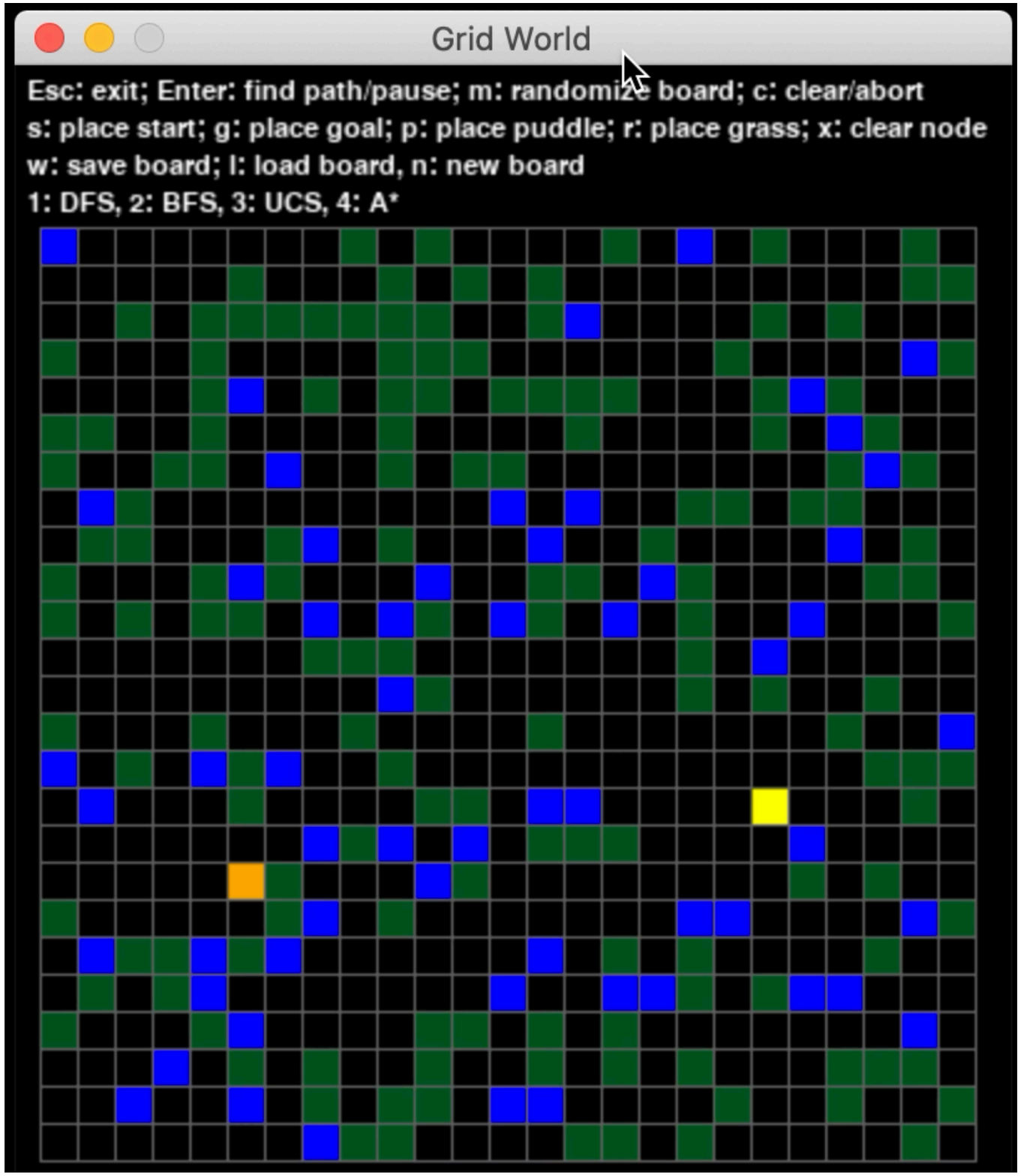
Analysis of UCS: Separation

- Lemma: At the end of each iteration of UCS, every acyclic path from any explored node to any unexplored node passes through at least one node in the frontier.






+5
+inf
+1


Grid World
Esc: exit; Enter: find path/pause; m: randomize board; c: clear/abort
s: place start; g: place goal; p: place puddle; r: place grass; x: clear node
w: save board; l: load board, n: new board
1: DFS, 2: BFS, 3: UCS, 4: A*

Analysis of UCS: Separation

- Lemma: At the end of each iteration of UCS, every acyclic path from any explored node to any unexplored node passes through at least one node in the frontier.
- How to prove this?
  - Induction over the steps of the algorithm.

Analysis of UCS: Optimality

- When any node is expanded, UCS has found the optimal path to it (i.e., with minimum cost).

Proof: Suppose when node n is expanded, UCS has not found the optimal path to it. Then there exists another path to n that has lower cost. But by the Lemma, that other path must passes through at least one node in the current frontier. So…

Properties of UCS

- Completeness: Yes, except for pathological cases (actions that do nothing and no cost)

- Optimality: uniform-cost search expands nodes in order of their optimal path cost.

- Complexity: UCS does not care about number of steps, only cost! So it may need to explore more than the depth of the goal.

Uninformed vs. Heuristic

DFS, BFS, UCS are all uninformed search: you don't know where you are unless you happen to hit the goal.

Uninformed search is not the way to go for any slightly interesting problem.

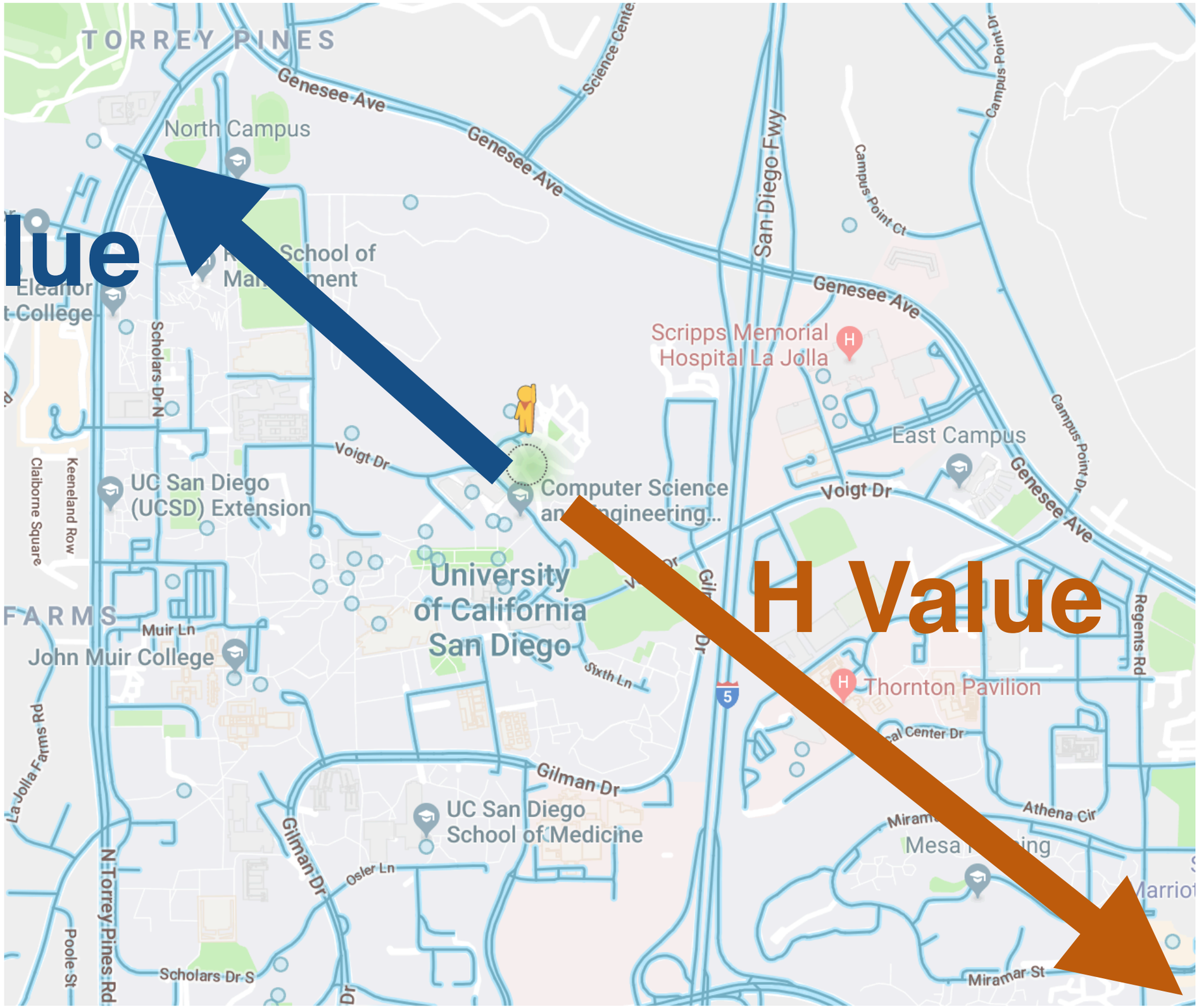
Informed Search: A*

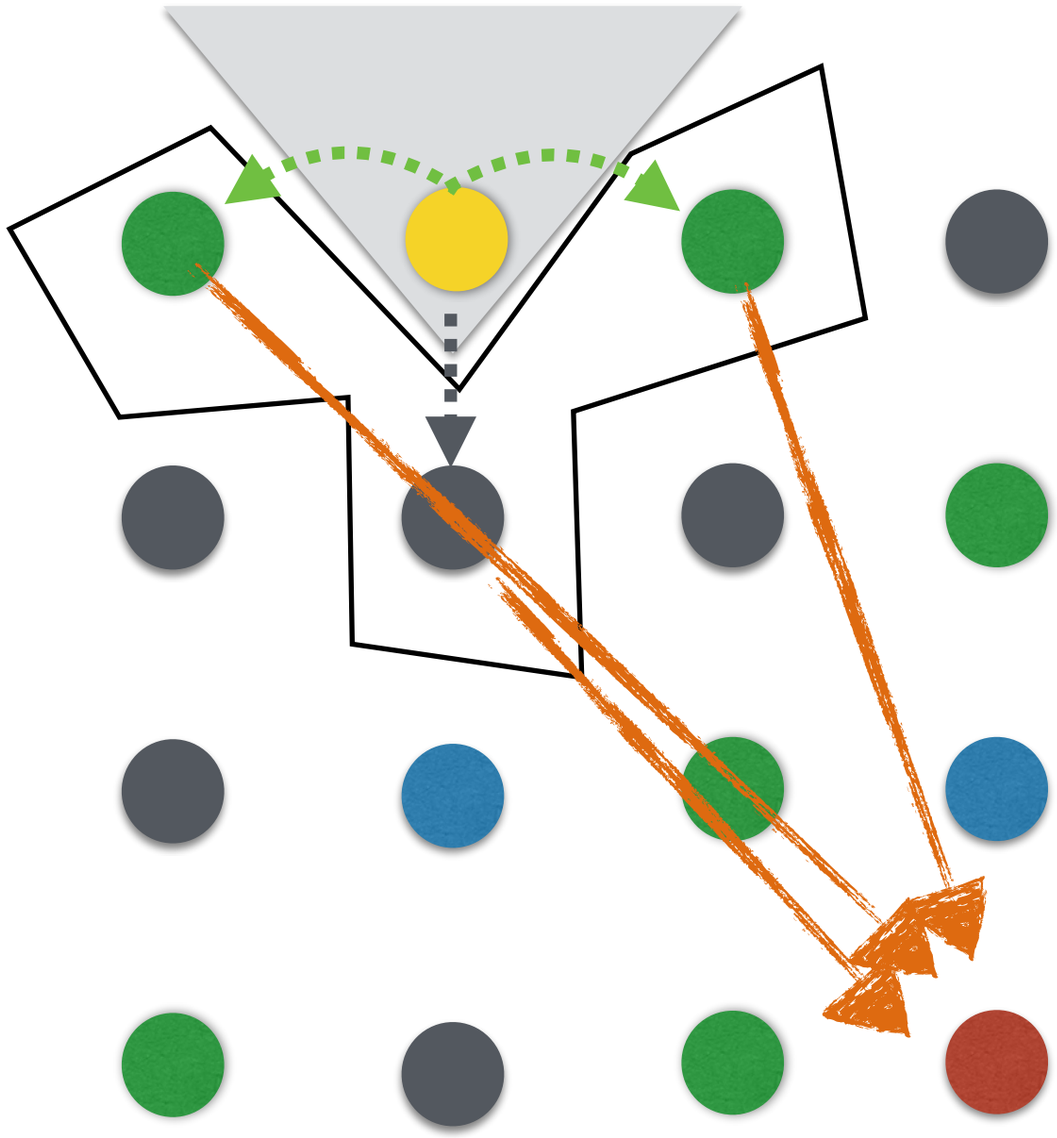
- What if you can measure how close you are to the goal?
  - This measure is always nonnegative and is zero at the goal.
  - Euclidean distance. Manhattan distance.

- Do exactly like UCS, but use a new cost:

$$F = G + H$$

Real cost + Heuristic cost


G Value
H Value
TORREY PINES
Genesee Ave
North Campus
School of
Eleanor
t-College
Scholars Dr N
Voigt Dr
UC San Diego
(UCSD) Extension
Computer Science
San Diego Fwy
Campus Point Dr
Campus Point Ct
Scripps Memorial
Hospital La Jolla
East Campus
University
of California
San Diego
FARMS
Muir Ln
John Muir College
Sixth Ln
Gilman Dr
UC San Diego
School of Medicine
Osler Ln
Scholars Dr S
La Jolla Farms Rd
N Torrey Pines Rd
Poole St
Thornton Pavilion
Athena Cir
Miramar St
Regents Rd




+5
+inf
+1

A* is Almost Just UCS

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution, or failure

node ← a node with STATE = *problem*.INITIAL-STATE, ~~PATH-COST = 0~~ G+H

frontier ← a priority queue ordered by ~~PATH-COST~~, with *node* as the only element G+H

explored ← an empty set

loop do

if EMPTY?(*frontier*) **then return** failure

node ← POP(*frontier*) /* chooses the lowest-cost node in *frontier* */

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

add *node*.STATE to *explored*

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child ← CHILD-NODE(*problem*, *node*, *action*)

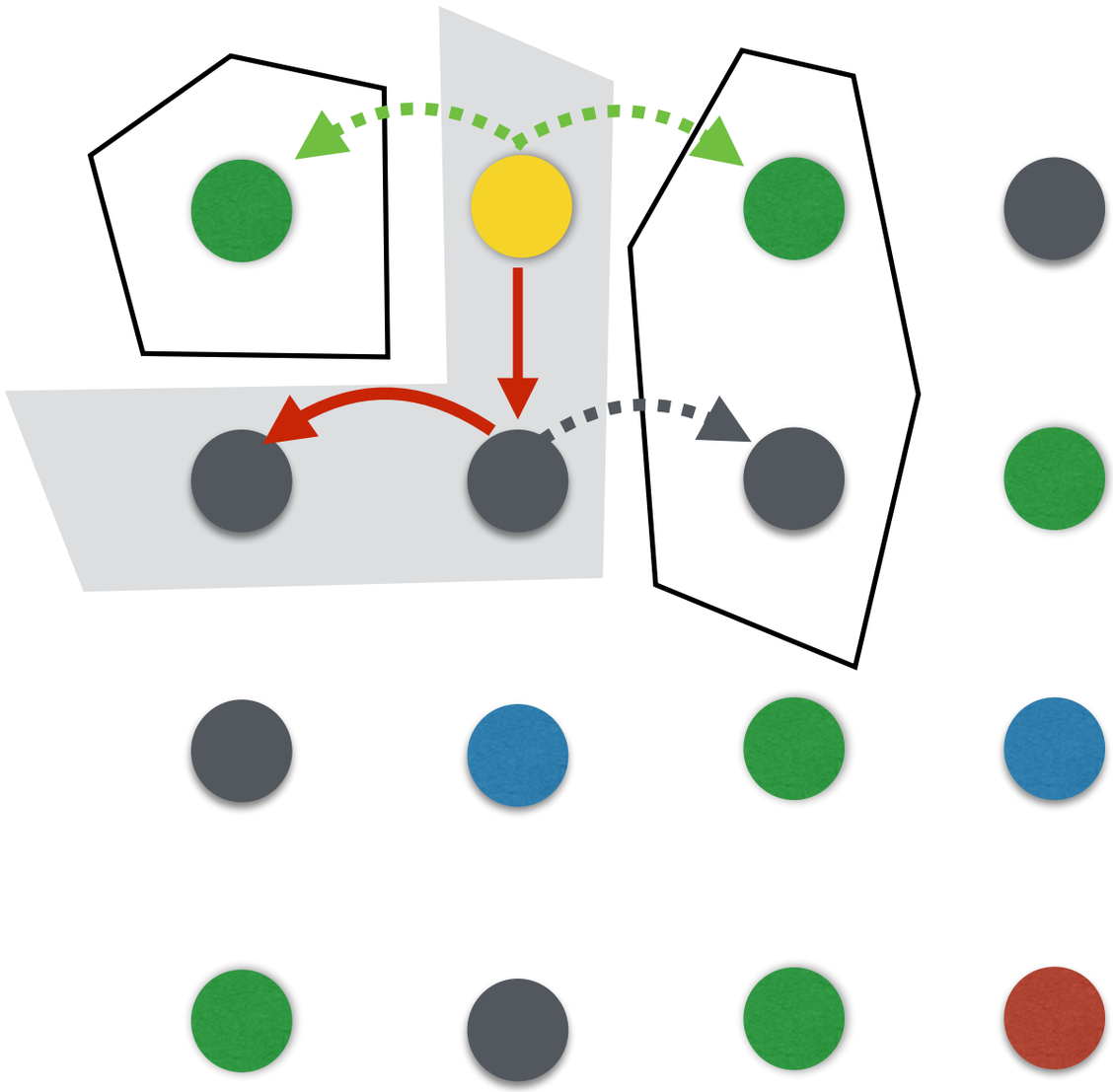
if *child*.STATE is not in *explored* or *frontier* **then**

frontier ← INSERT(*child*, *frontier*)

else if *child*.STATE is in *frontier* with higher ~~PATH-COST~~ G+H **then**

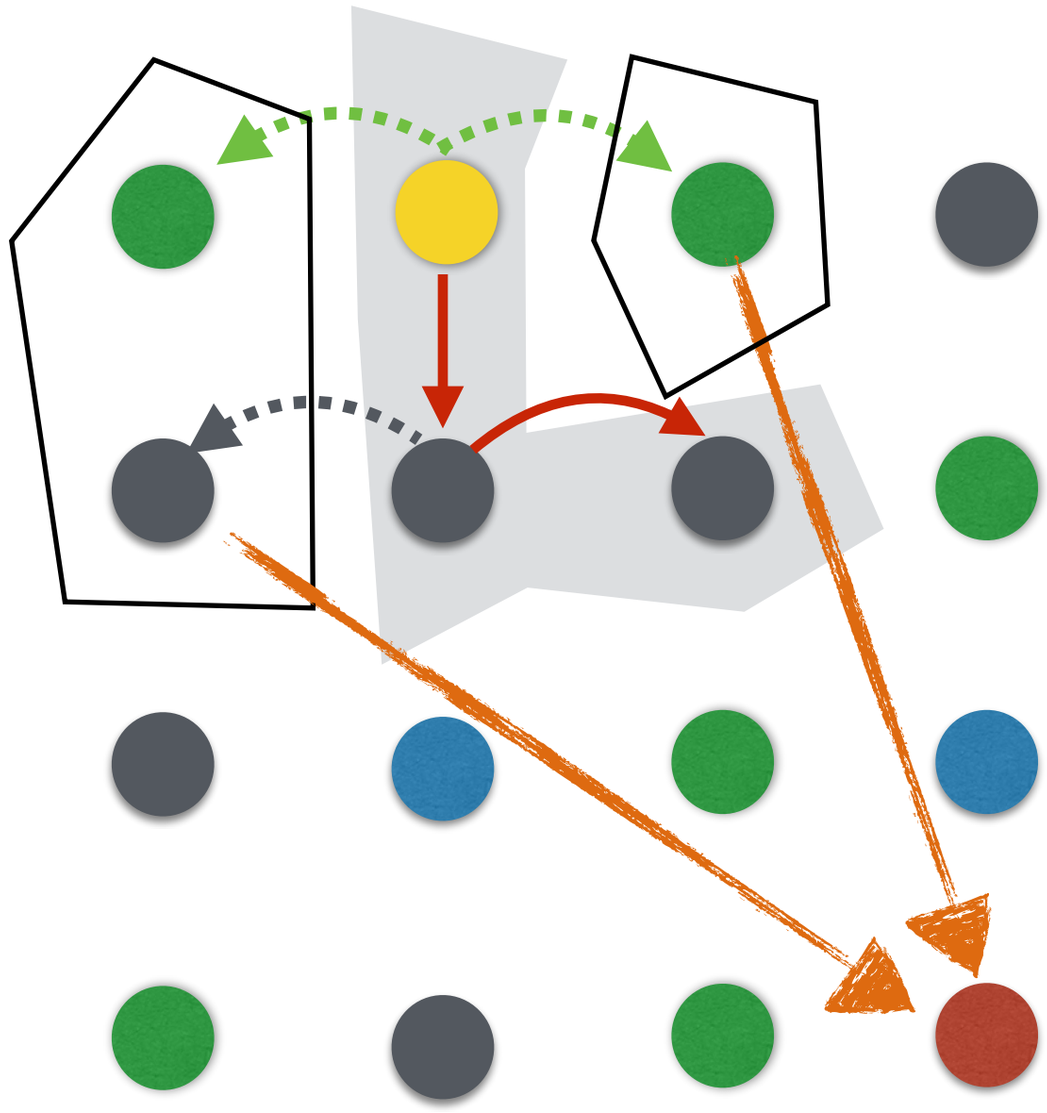
replace that *frontier* node with *child*

In UCS

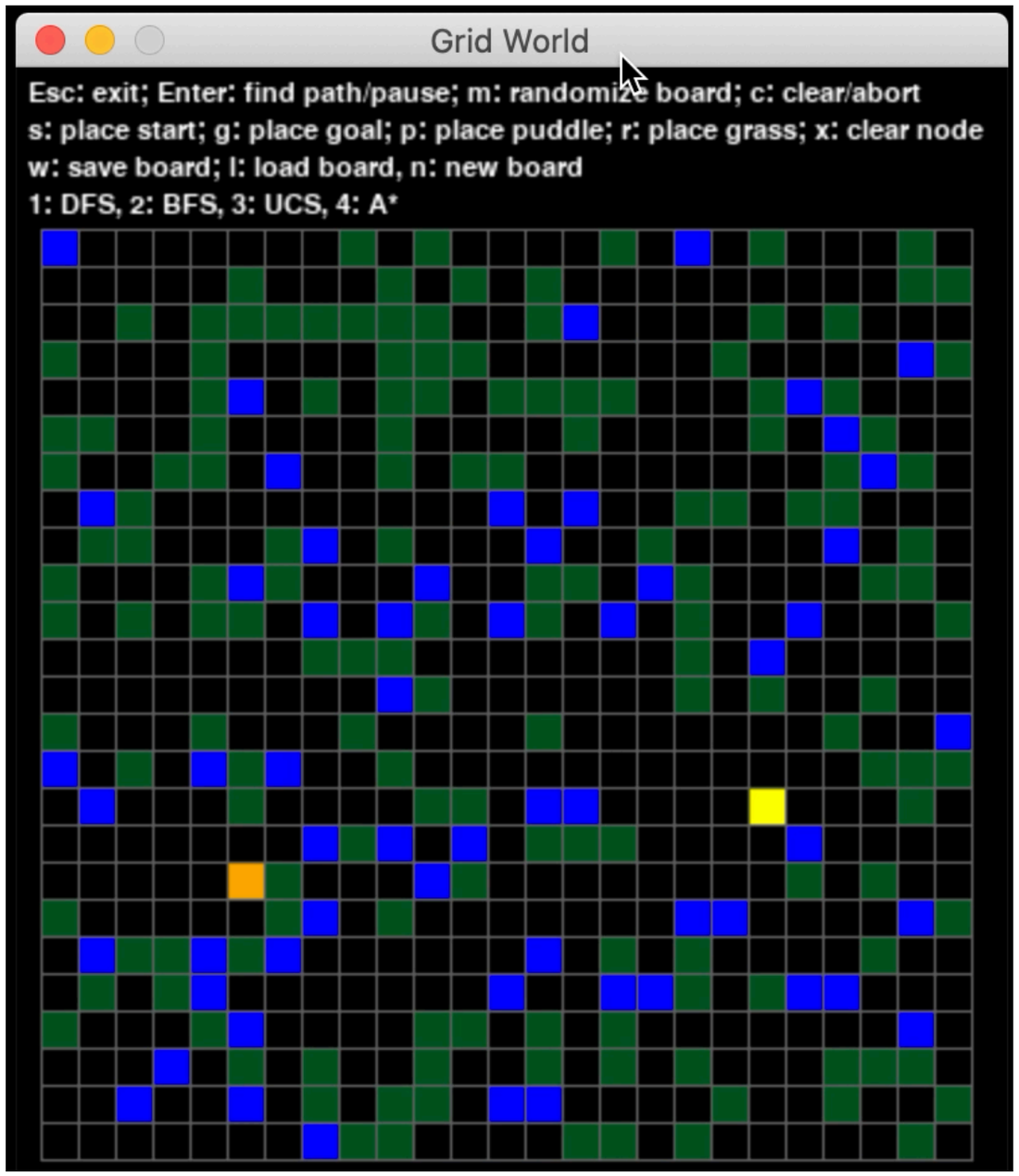


+5 +inf +1

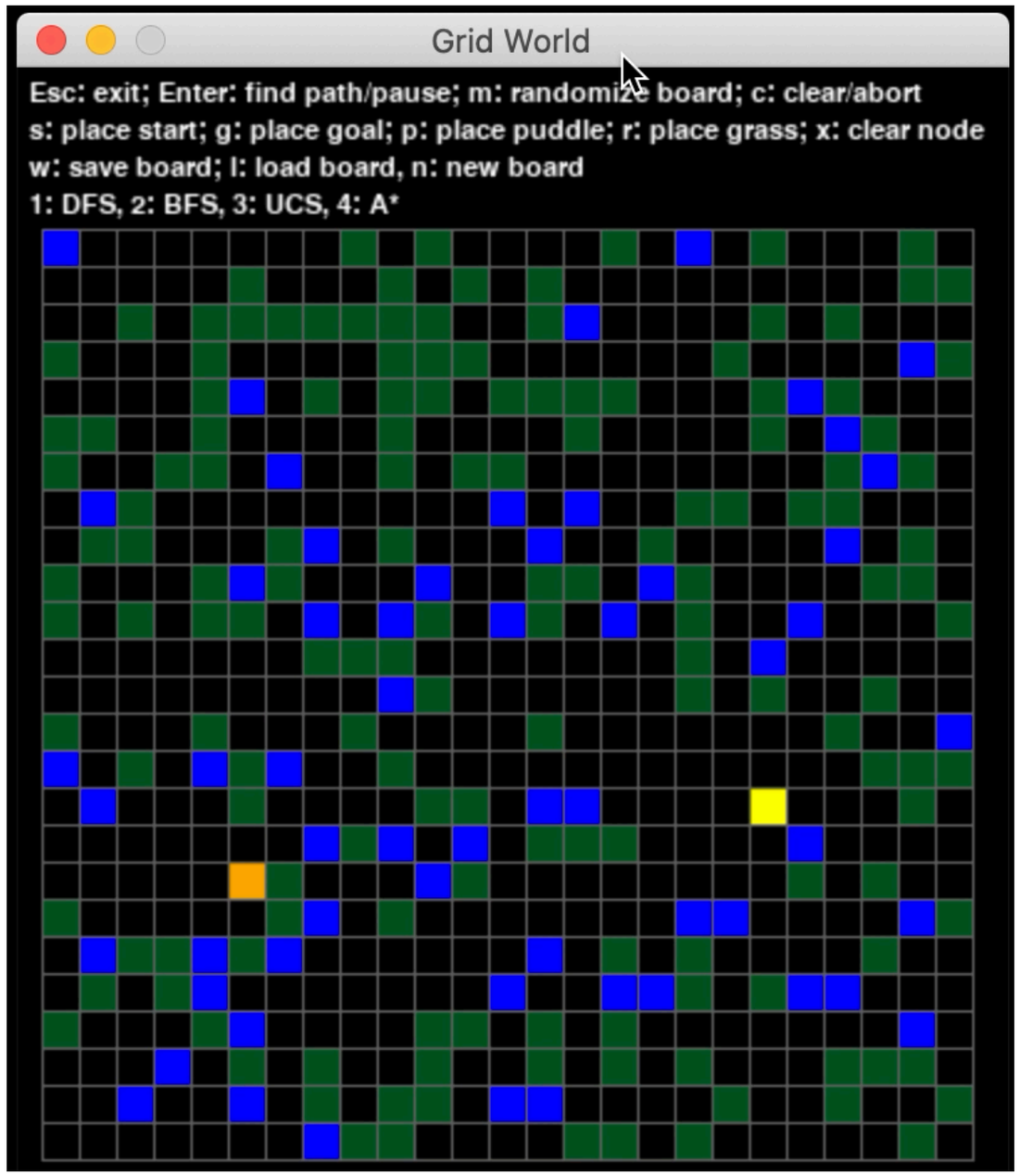
In A*



Does H affect optimality? How?


Grid World
Esc: exit; Enter: find path/pause; m: randomize board; c: clear/abort
s: place start; g: place goal; p: place puddle; r: place grass; x: clear node
w: save board; l: load board, n: new board
1: DFS, 2: BFS, 3: UCS, 4: A*

UCS


Grid World
Esc: exit; Enter: find path/pause; m: randomize board; c: clear/abort
s: place start; g: place goal; p: place puddle; r: place grass; x: clear node
w: save board; l: load board, n: new board
1: DFS, 2: BFS, 3: UCS, 4: A*

A*

Key requirements for good heuristics

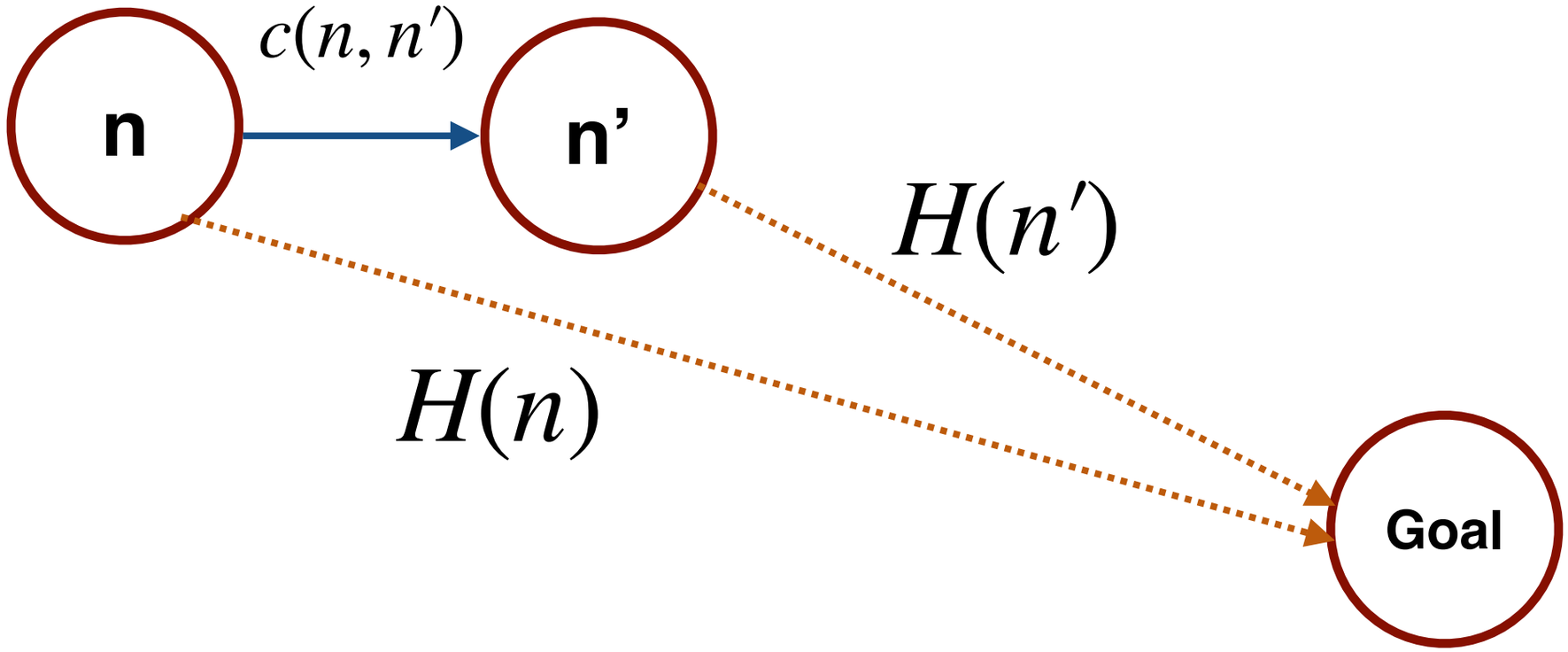
- Basic:
  - H(goal) = 0
  - H is always nonnegative

- When can A* find the optimal solution?
  - What if H is super small?
  - What if H is super large?

Analysis of A*: Optimality

- Consistency: Whenever n' is a child of n,

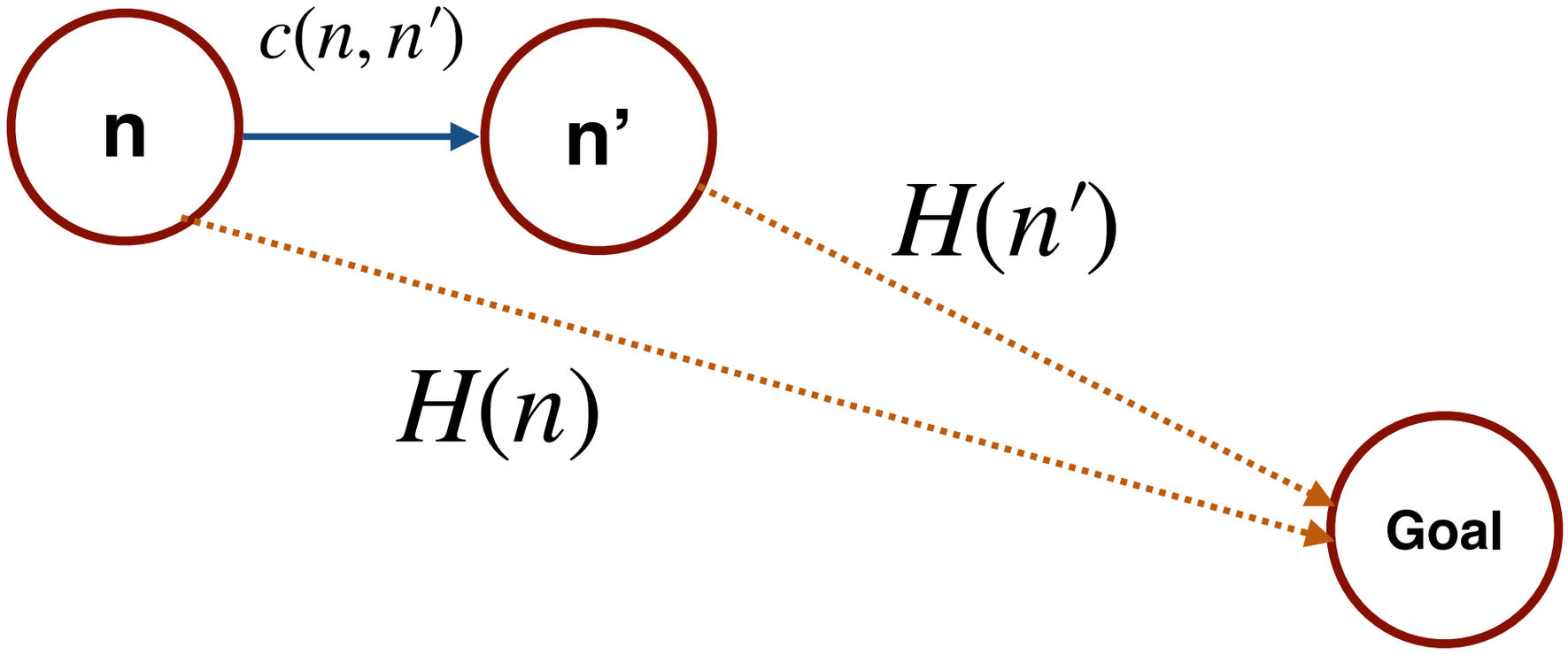
$$H(n) \leq H(n') + c(n, n')$$



Analysis of A*: Optimality

- Consistency: $H(n) \leq H(n') + c(n, n')$

- Euclidean distance and Manhattan distance
  (when each step costs no less than 1)



Analysis of A*: Optimality

- Consistency (never give contradictory predictions): Whenever n' is a child of n,

$$G(n) + H(n) \leq H(n') + c(n, n') + G(n)$$

Analysis of A*: Optimality

- Consistency (never give contradictory predictions): Whenever n' is a child of n,

$$\frac{G(n)+H(n) \leq H(n') + c(n,n') + G(n)}{F(n)}$$

Analysis of A*: Optimality

- Consistency (never give contradictory predictions): Whenever n' is a child of n,

$$\frac{G(n) + H(n) \leq H(n') + c(n, n') + G(n)}{F(n')}$$

Analysis of A*: Optimality

- Consistency (never give contradictory predictions): Whenever n' is a child of n,

$$\frac{G(n) + H(n) \leq H(n') + c(n, n') + G(n)}{F(n) \leq F(n')}$$

Analysis of A*: Optimality

$$F(n) \leq F(n')$$

- Now recall that A* is exactly the same algorithm as UCS, using F as the cost function.

- So from optimality of UCS, we know A* always finds the path with lowest F cost.

- Now, what is F at the goal node??

Analysis of A*: Optimality

Theorem

A* always finds the optimal path (minimum real cost) when the heuristic function is consistent.

Analysis of A*:

- **Consistency** (never give contradictory predictions): Whenever n' is a child of n,

$$H(n) \leq H(n') + c(n, n')$$

- Consistency implies Admissibility:

$$H(n) \leq \min_{\pi} \text{Cost}(\pi(n, \text{goal}))$$

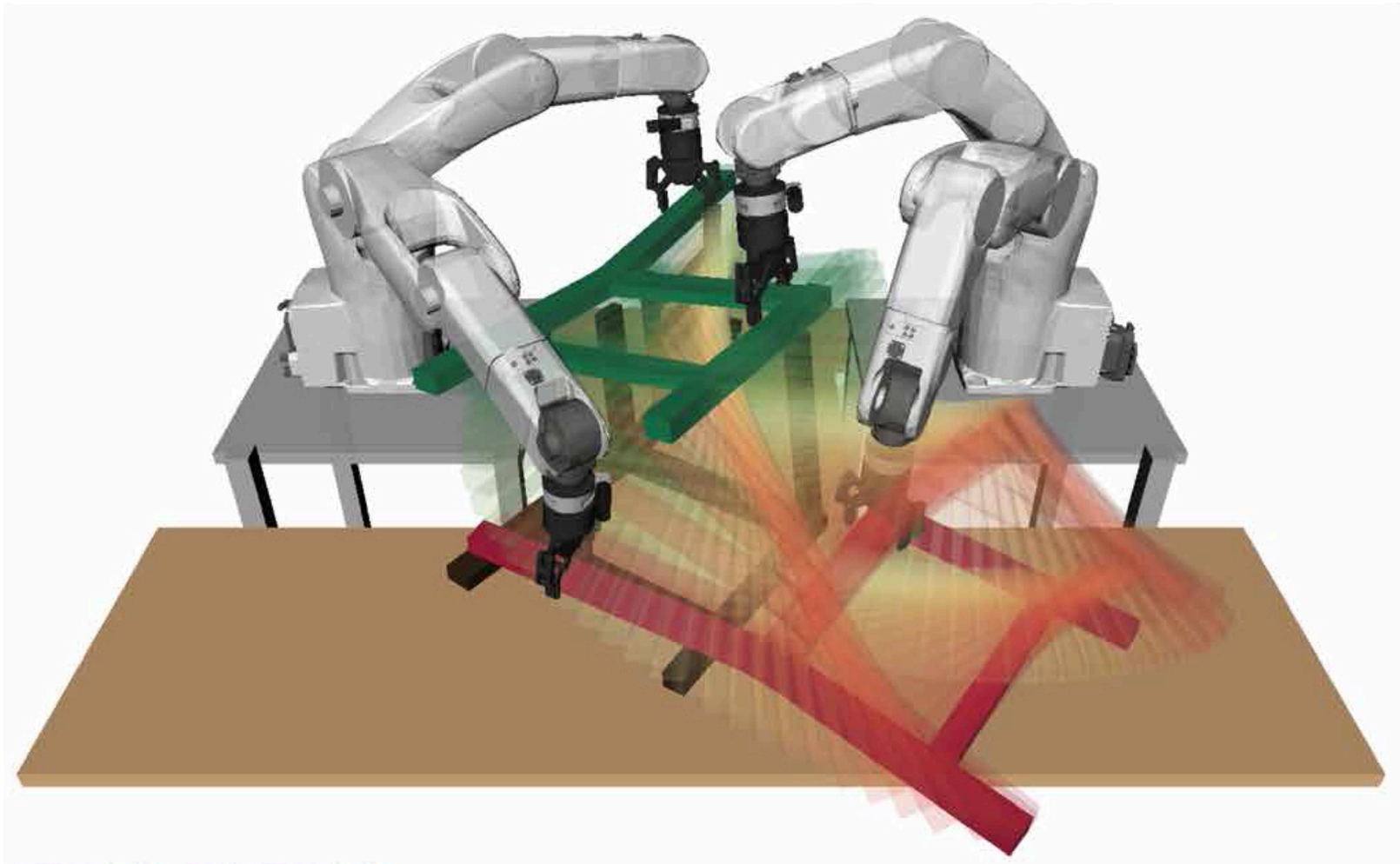
But in the end, what is good?

Trade off between optimality and speed.

- Big H values: faster search
- Small H values: more optimal

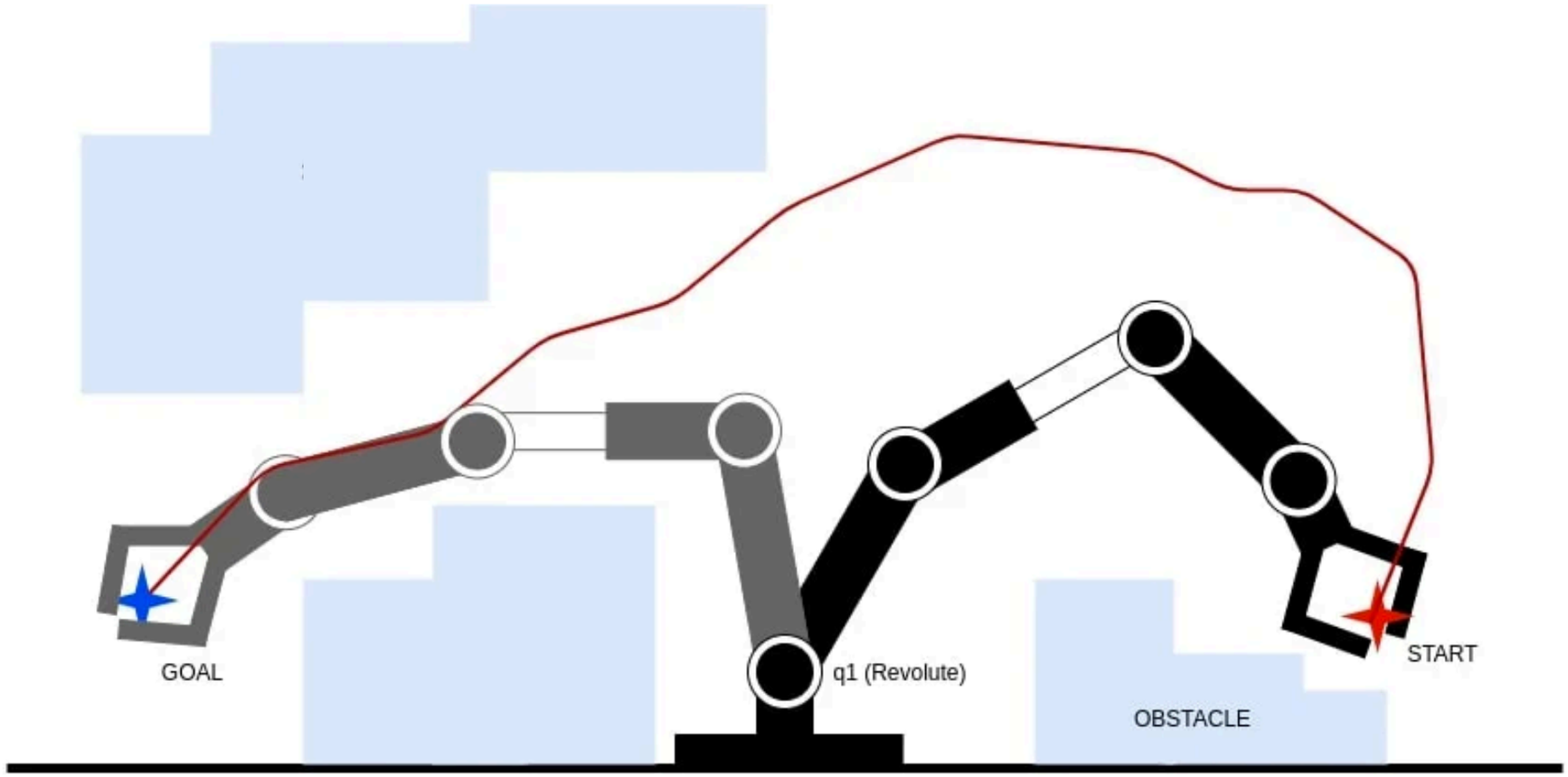
This is where science becomes art. Learning can often help you.

Motion Planning (possible project)



Motion Planning

- What if there is no explicit graph?



General Path Finding

- Configuration space: high-dimensional
- Work space: at most 3D
- Find a path (sequence of states in the configuration space) from start to goal such that each segment of the path is collision-free
  - Collision checking can be very expensive

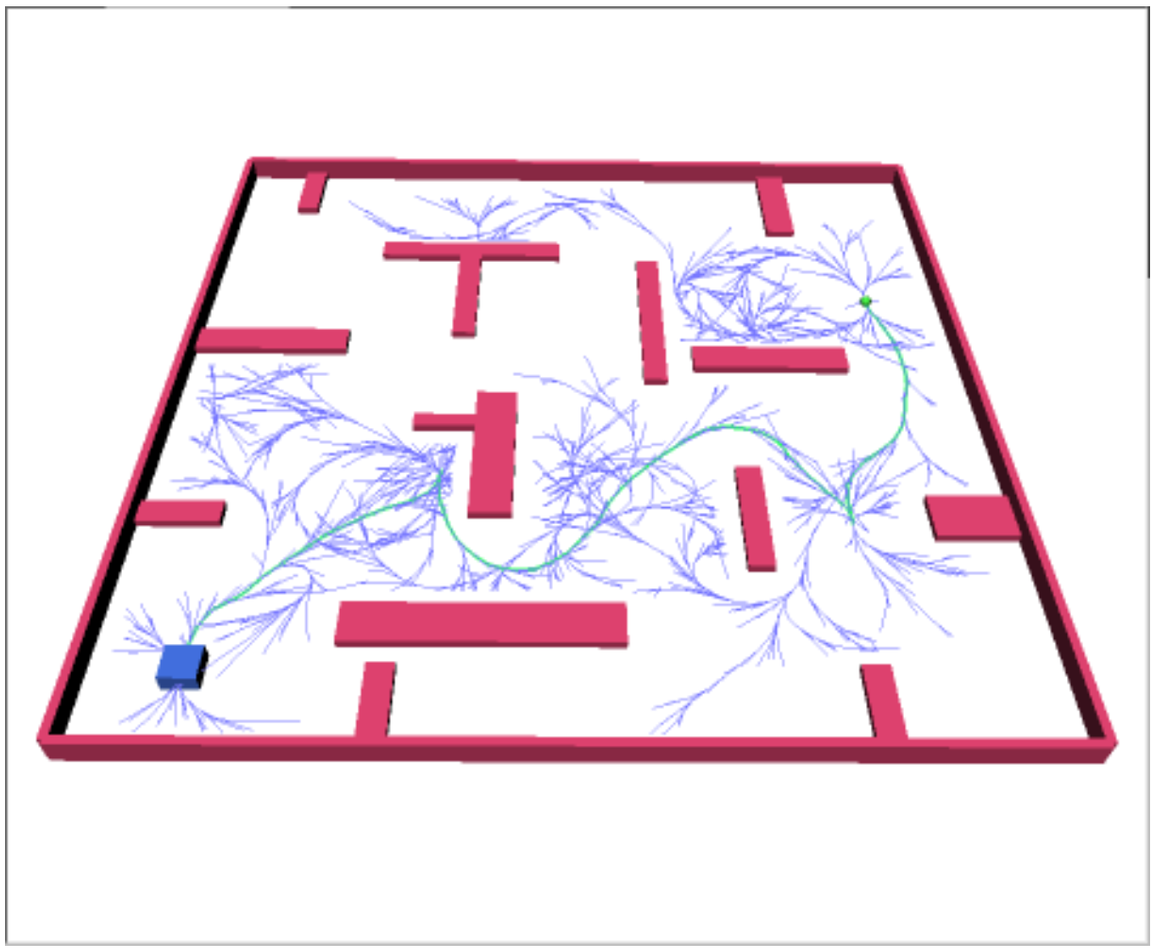
General Path Finding

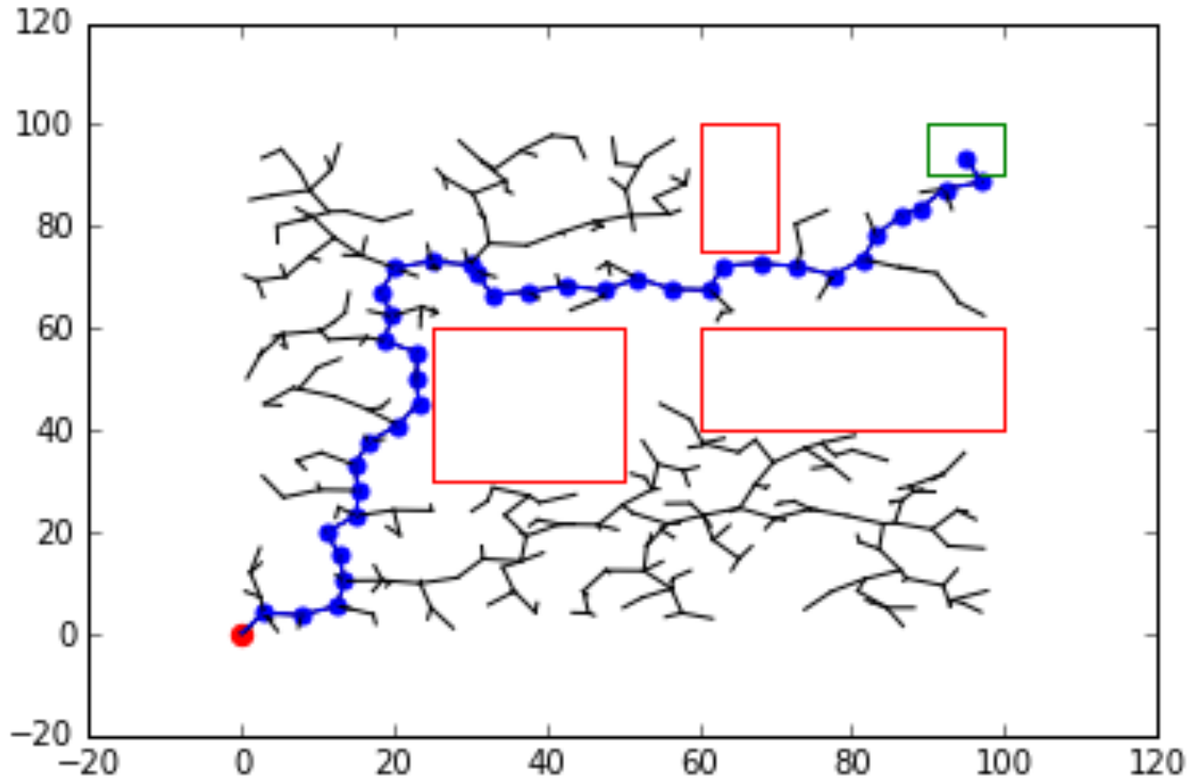
- Configuration space $C \subseteq \mathbb{R}^n$, $C = C_o \cup C_f$
  - Obstacles C_o
  - Free space C_f
- Start $s_0 \in C_f$ and goal $s_g \in C_f$
- Find state sequence $s_0, \ldots, s_k$ such that $s_k = s_g$ and the edge $E(s_i, s_{i+1}) \subseteq C_f$ for all $0 \leq i < k$

Rapidly Exploring Random Trees (RRT)

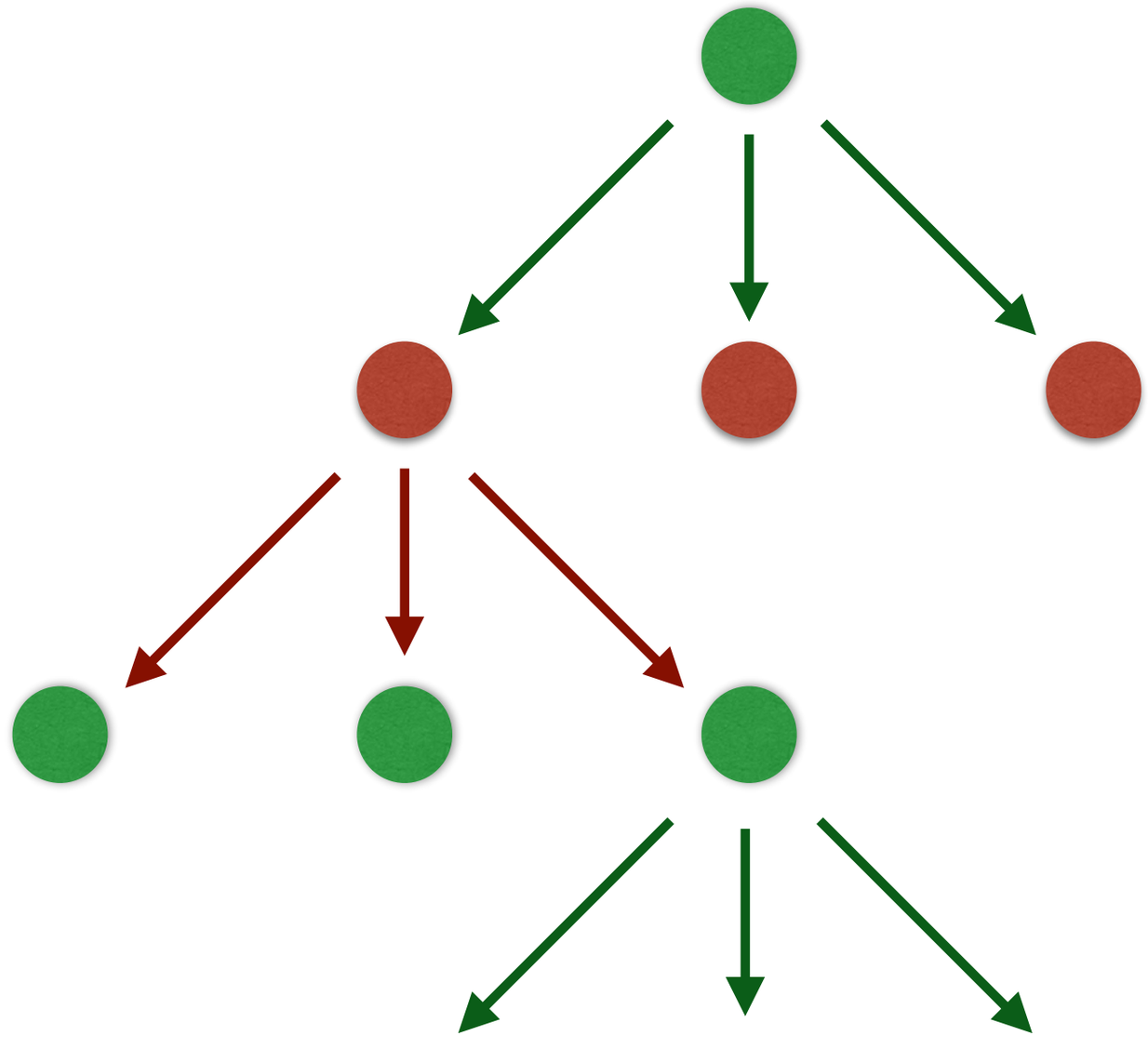
- Grow a tree in the free space
  - In each iteration, randomly sample a state from the free space.
  - Find the node in the current tree nearest to the sampled state, and grow the tree at that node by a small step
  - Periodically check if the goal is a small distance away from any node in the tree

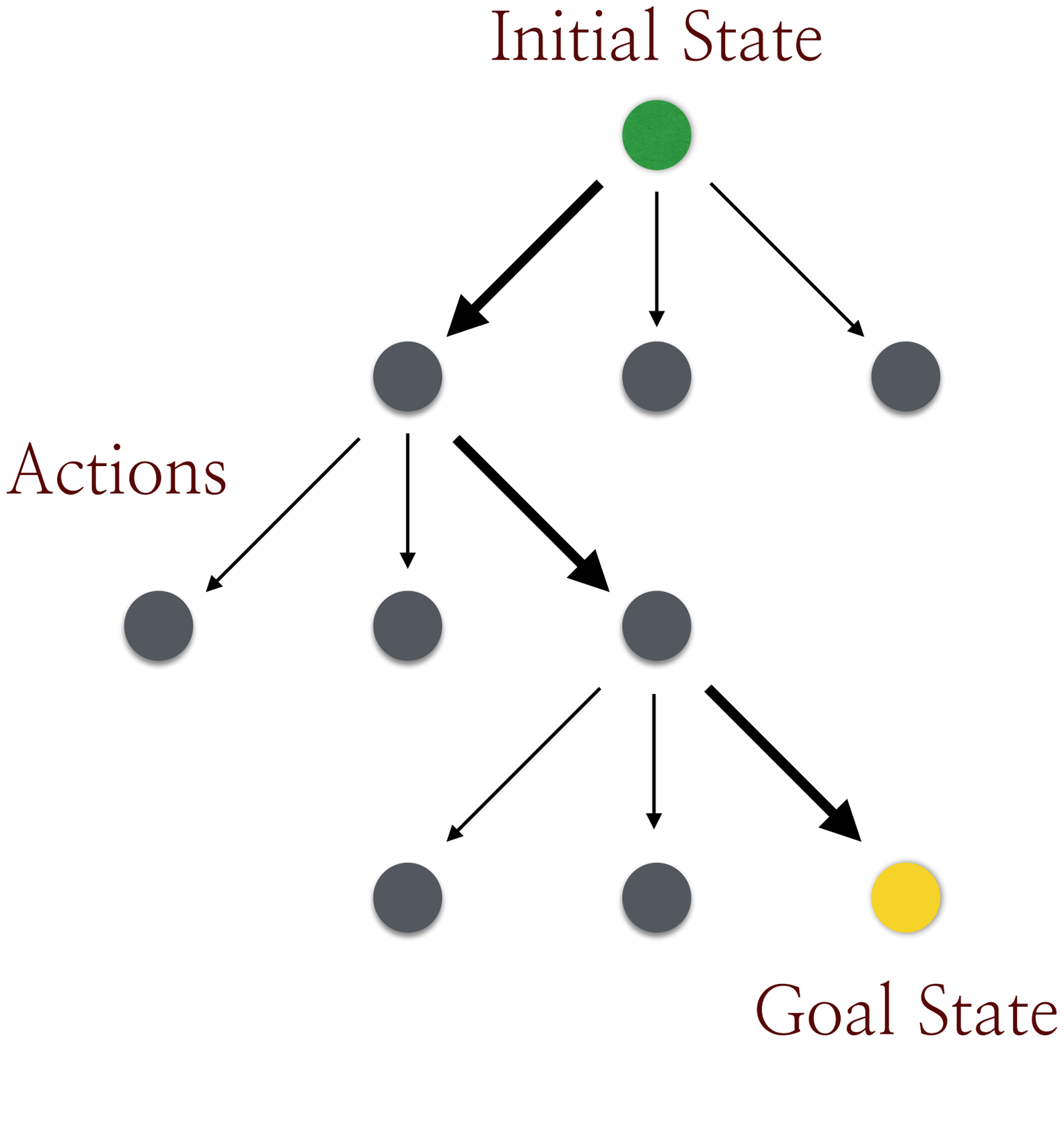
Rapidly Exploring Random Trees (RRT)

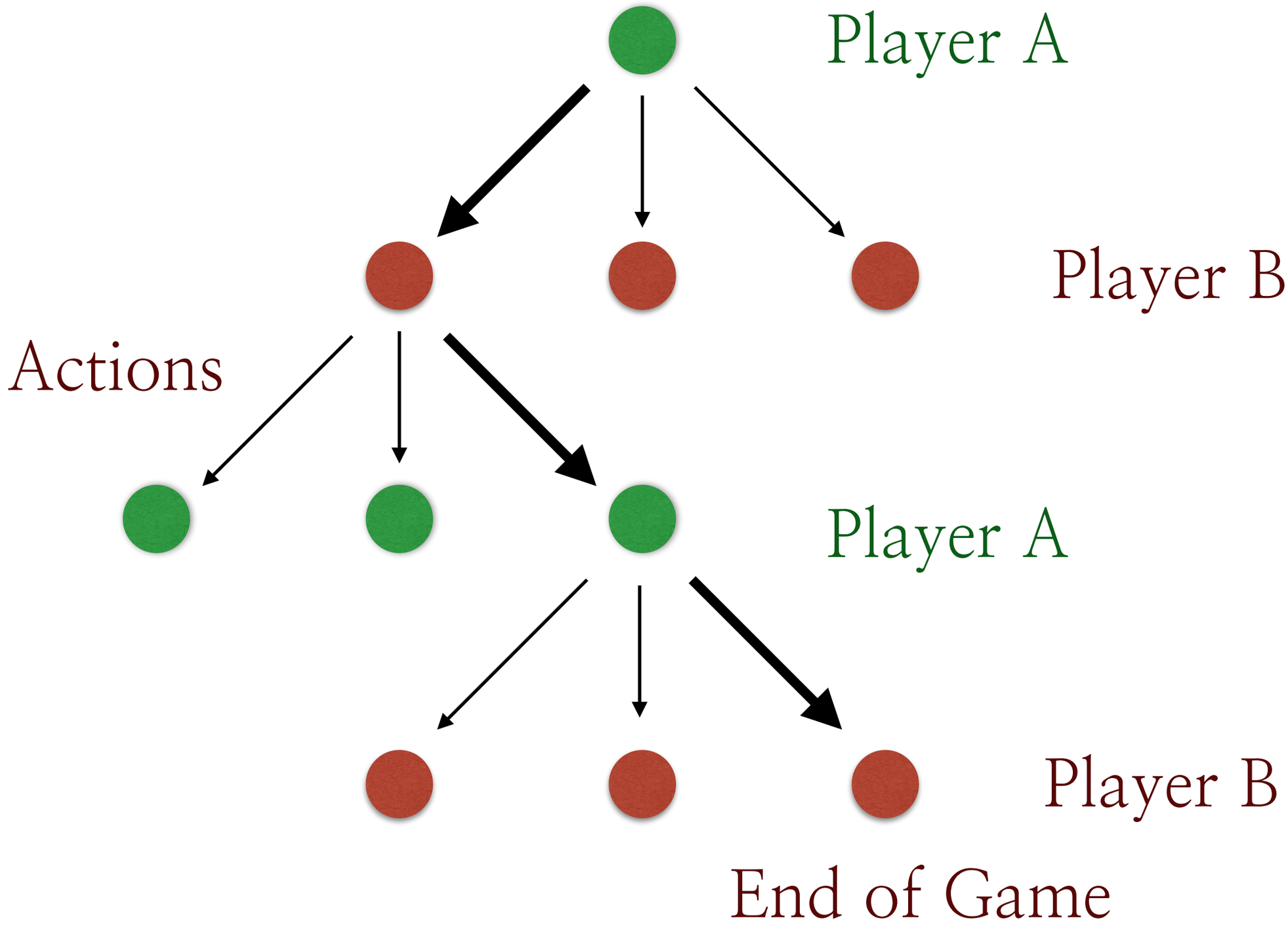




Adversarial Search




Initial State
Actions
Goal State


Player A
Player B
Actions
Player A
Player B
End of Game

C wants max, R wants min, C picks first

	C1	C2
R1	7	5
R2	2	10

C wants max, R wants min, C picks first

	C1	C2
R1	7	5
R2	2	10

Moves: (C2, R1)

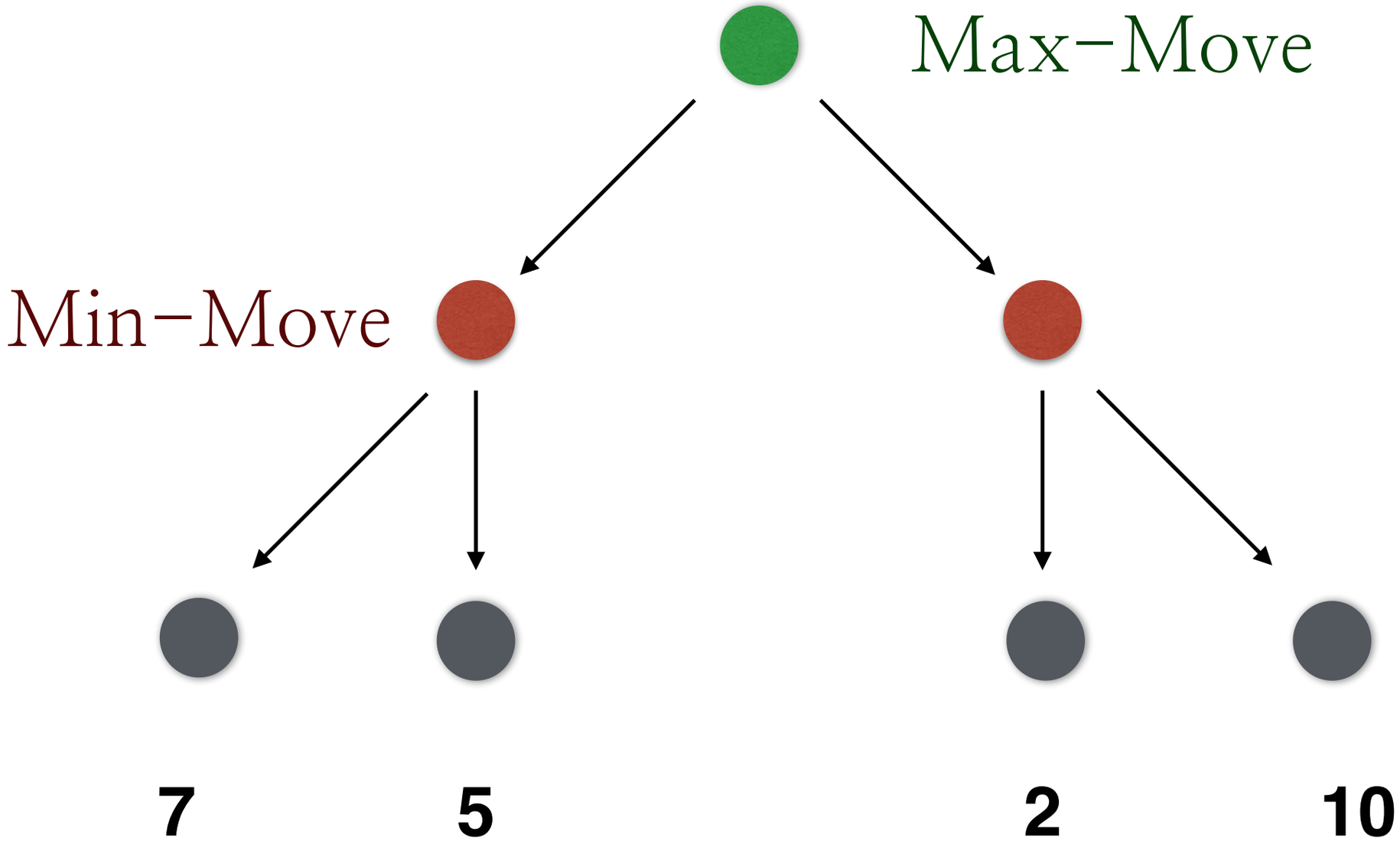
C wants max, R wants min, R picks first

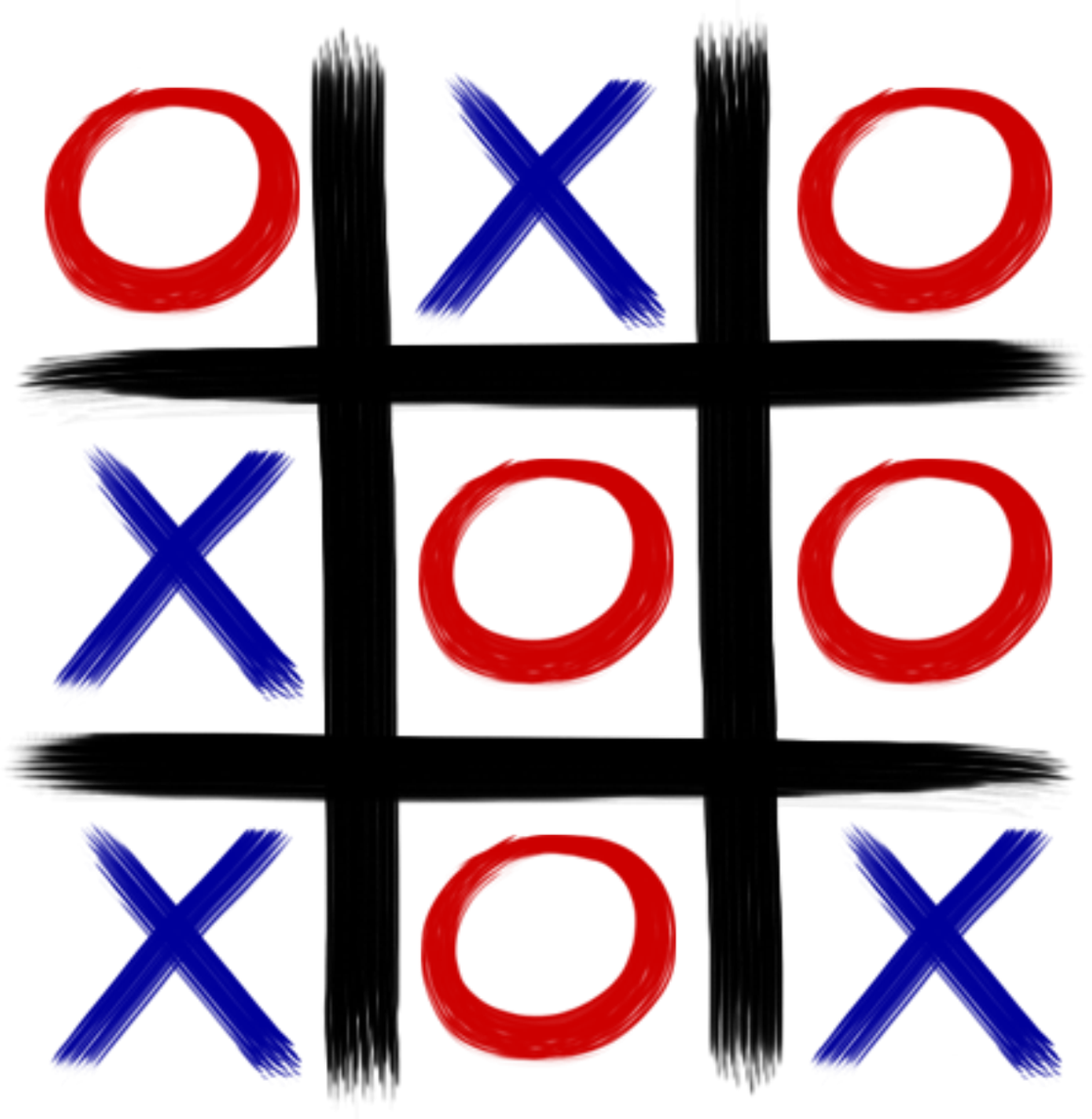
	C1	C2
R1	7	5
R2	2	10

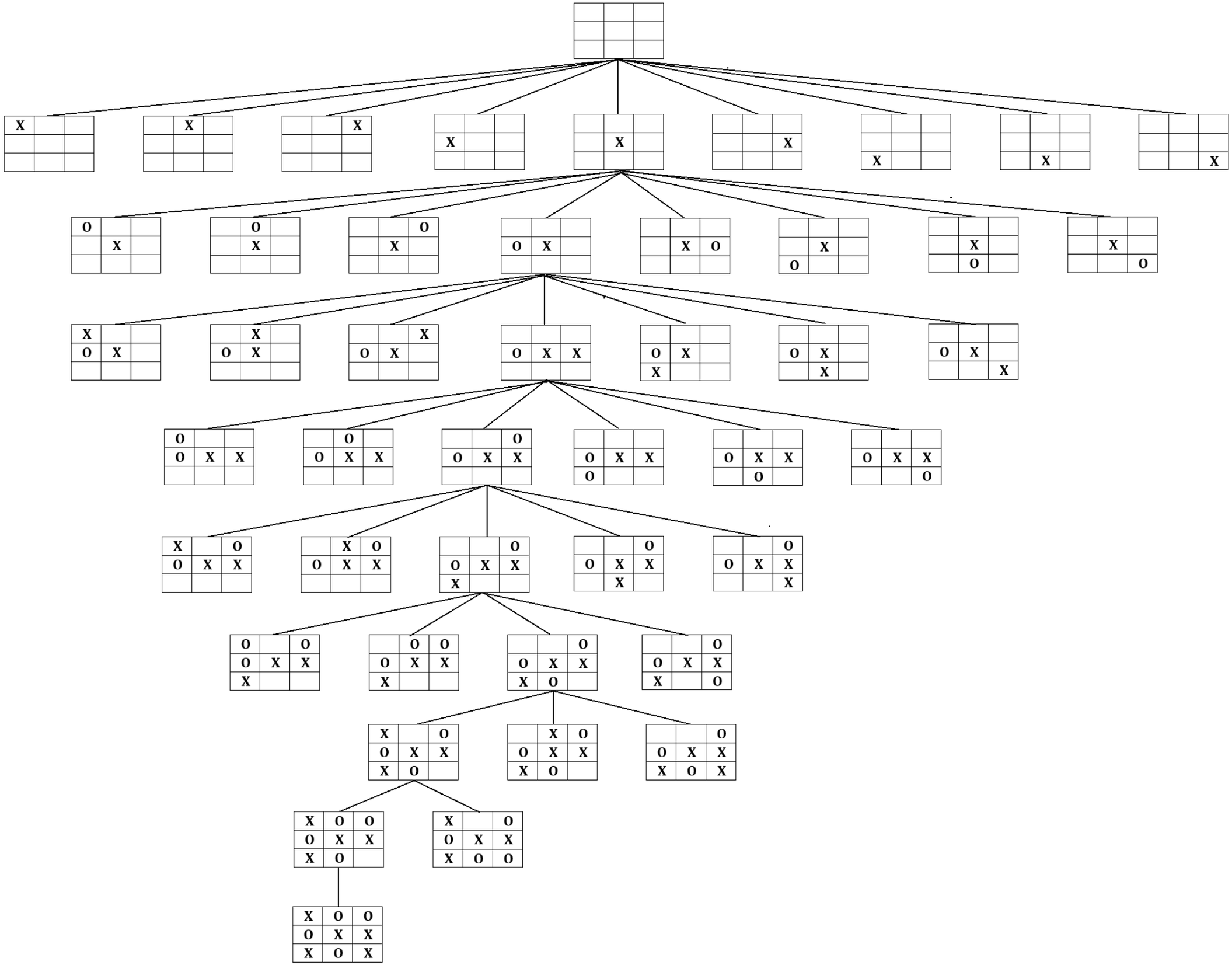
C wants max, R wants min, R picks first

	C1	C2
R1	7	5
R2	2	10

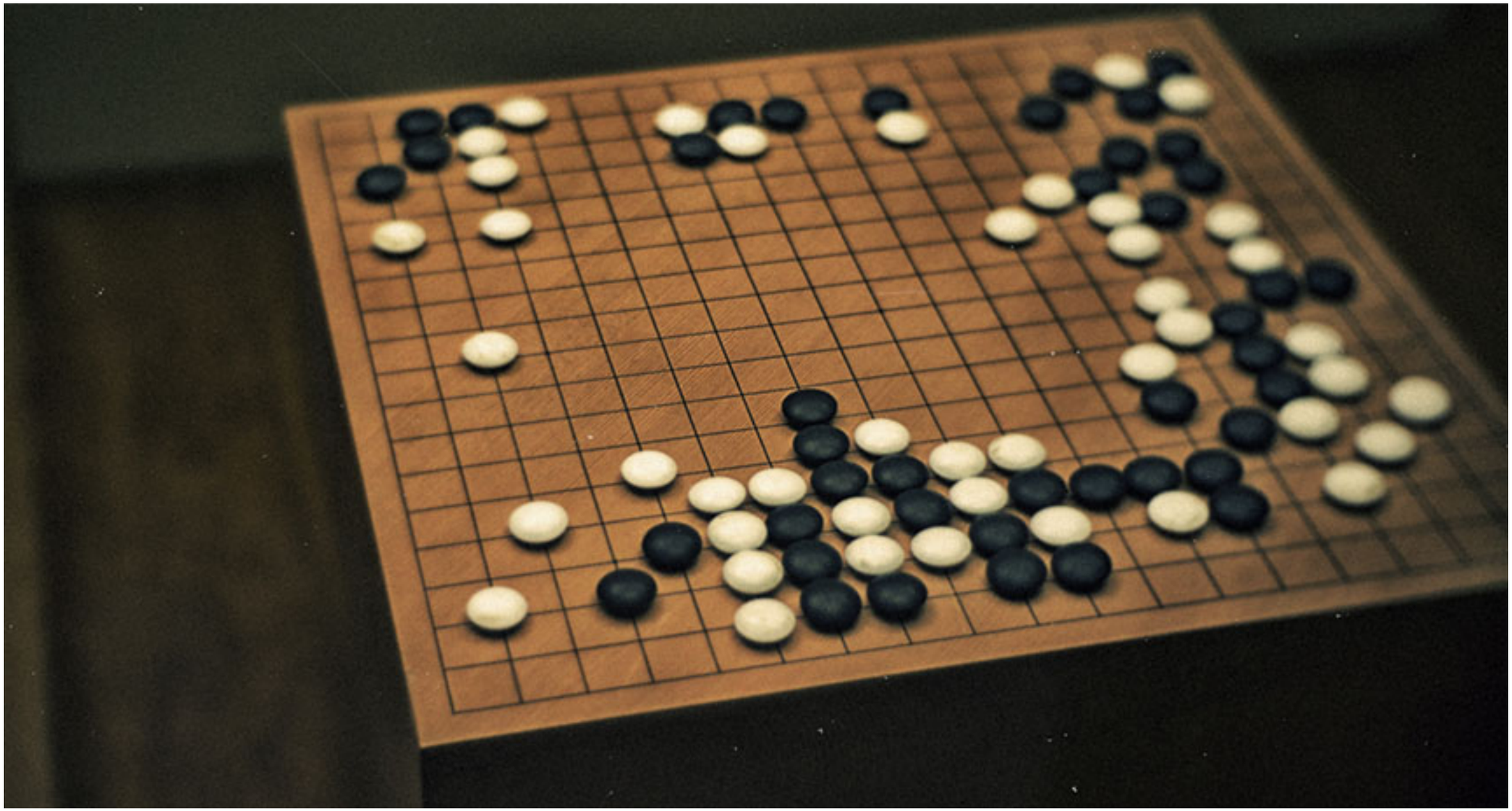
Moves: (R1, C1)


Max-Move
Min-Move
7
5
2
10



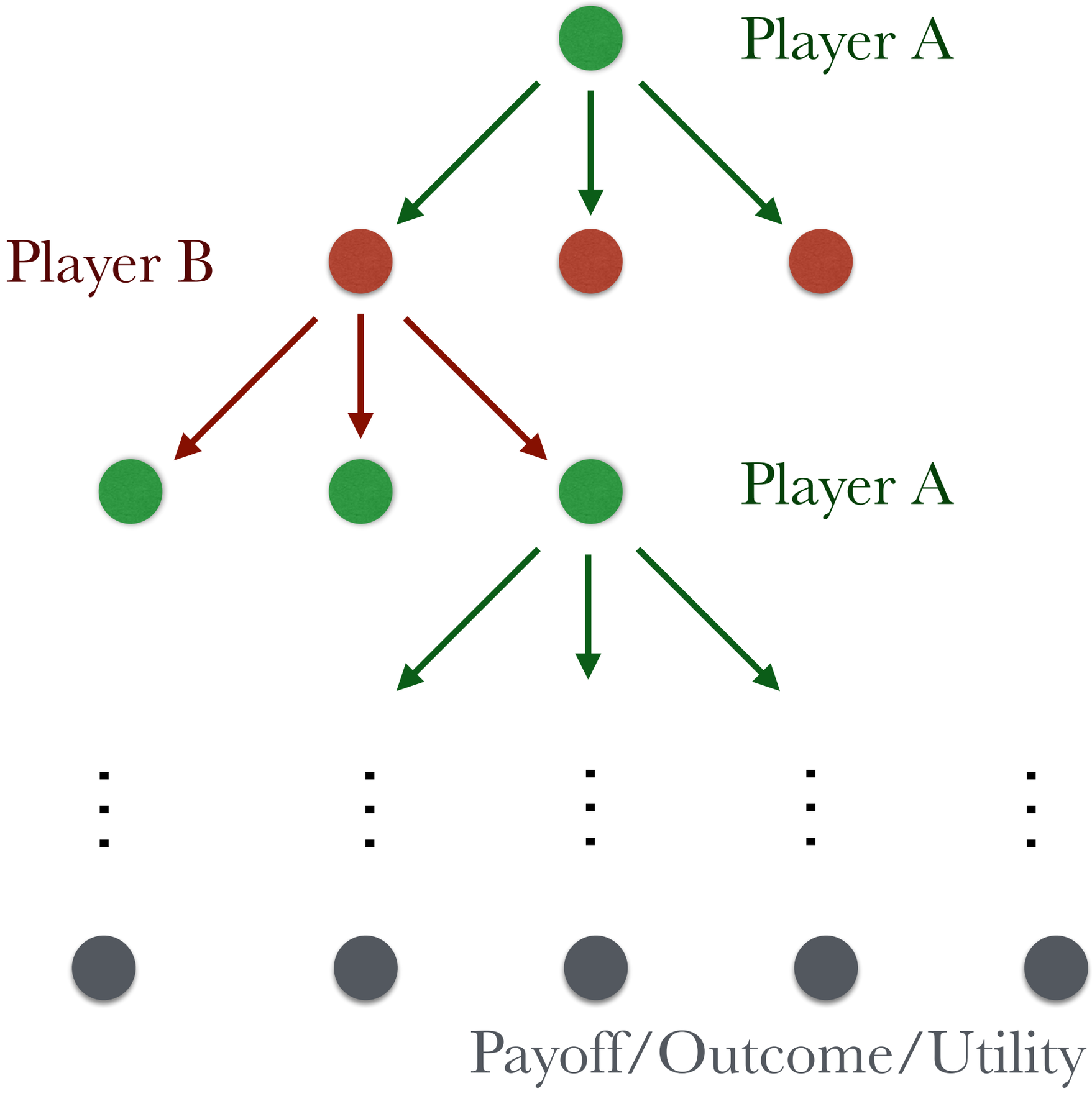


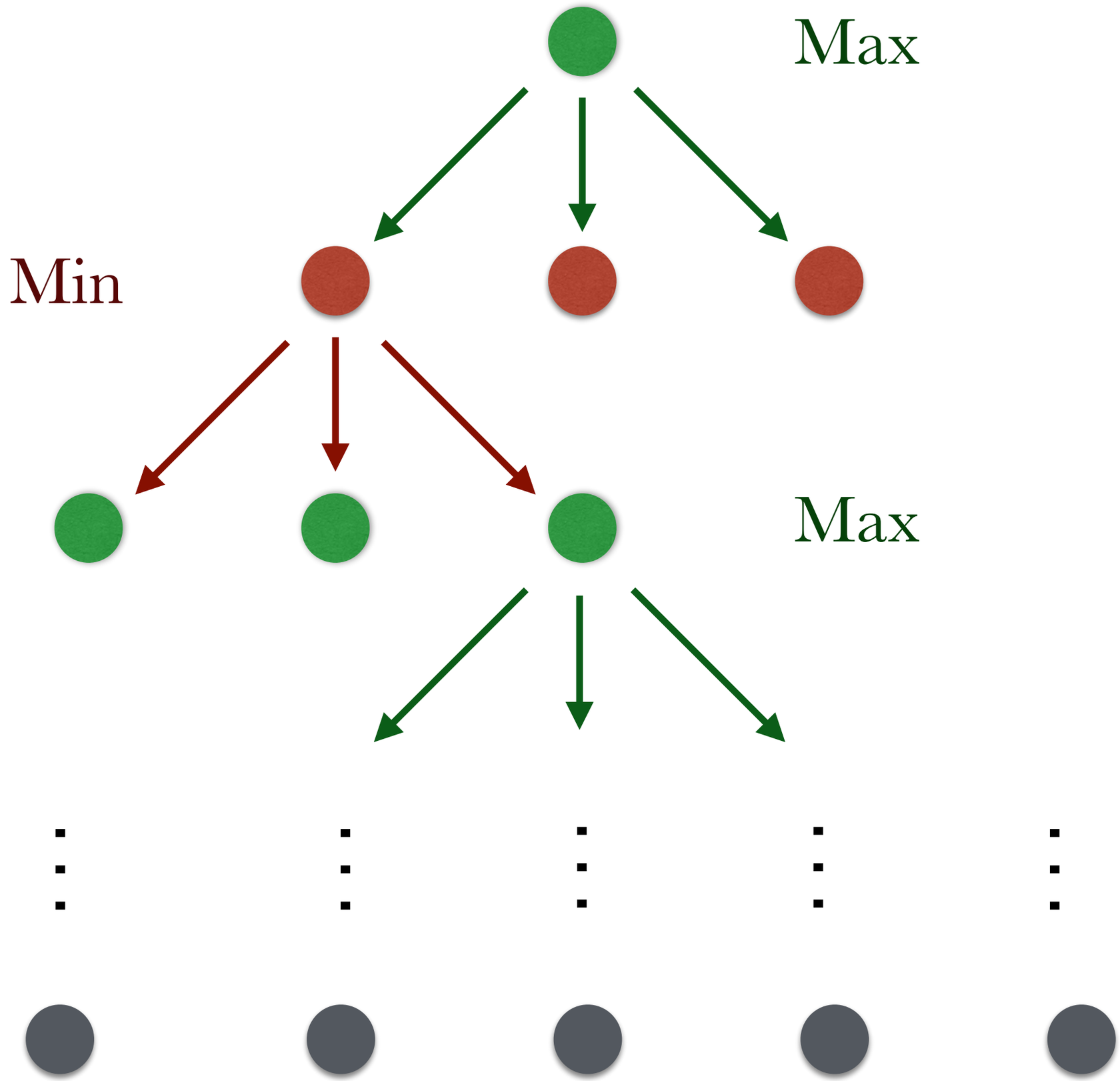
(Winner - X)

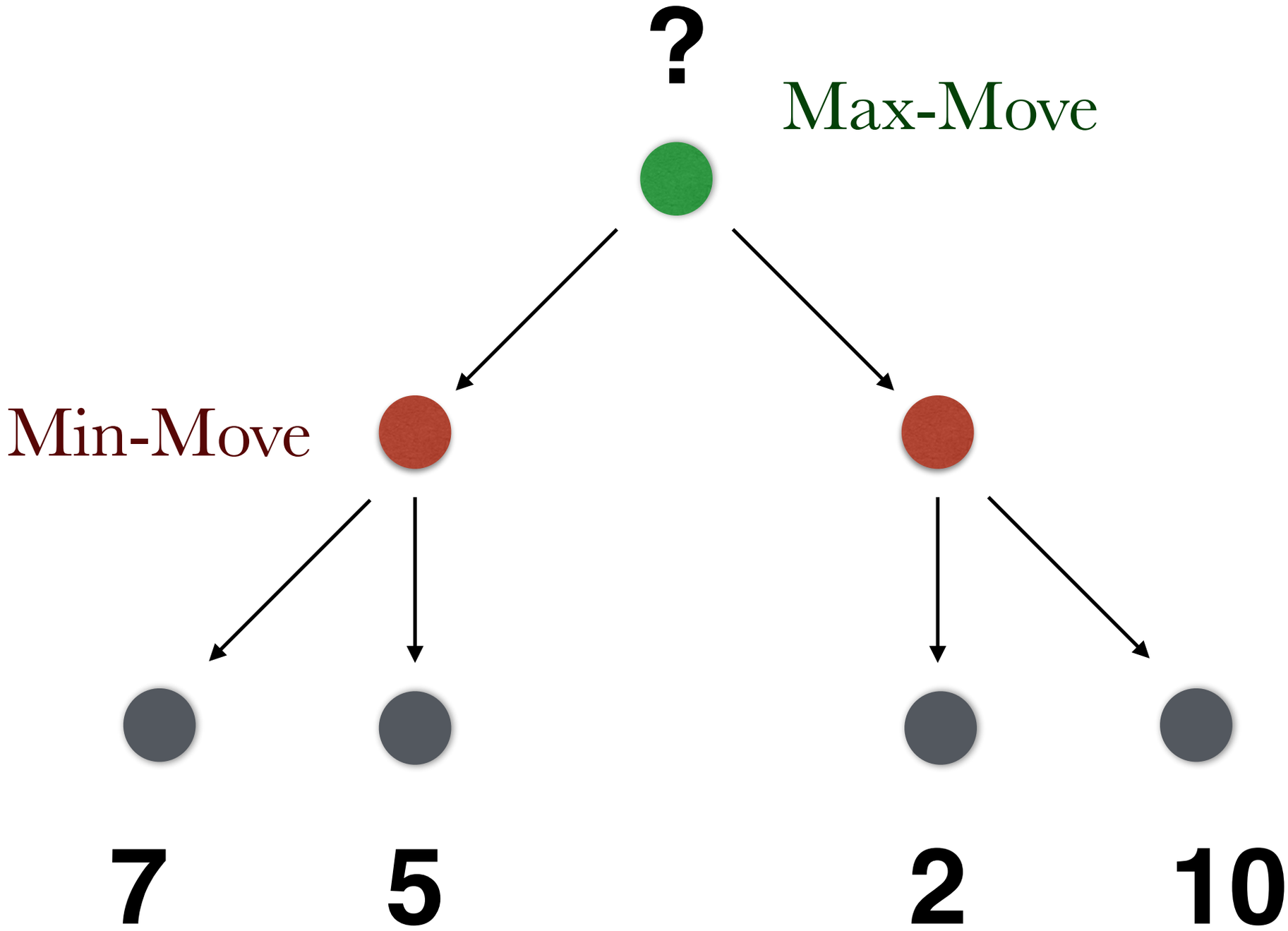


Basic Two-Player Zero-Sum Games

- Two players, take turns
- Actions: each player has a finite number of actions
- Finite length: all sequences of moves terminate
- Zero-Sum: strictly competitive, the gain of one player at the end is exactly the loss of the other player
- Perfect information


Player A
Player B
Player A
Payoff/Outcome/Utility


Max
Min
Max
Payoff/Outcome/Utility


?
Max-Move
Min-Move
7
5
2
10

Minimax Values

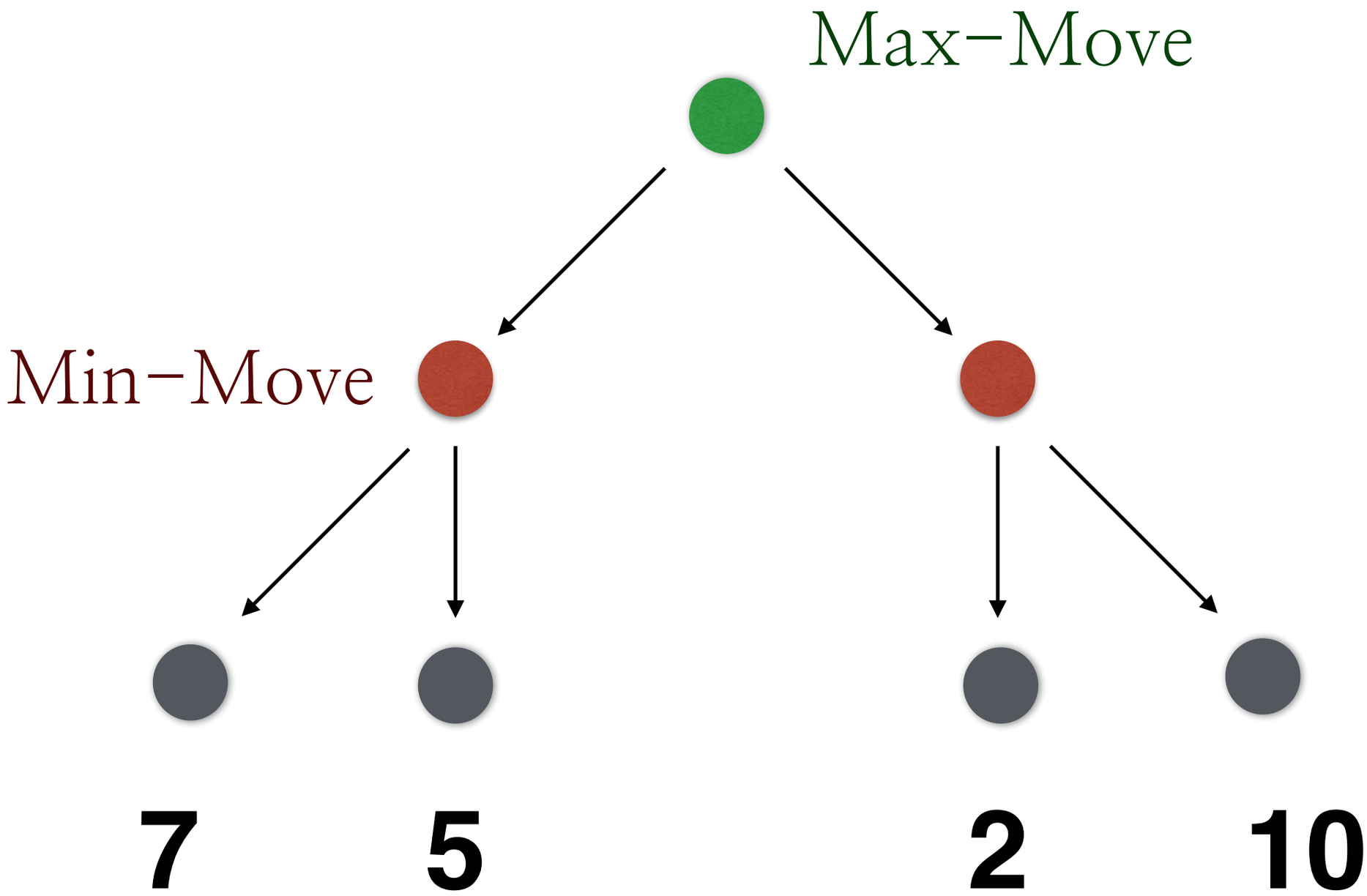
Minimax(node):

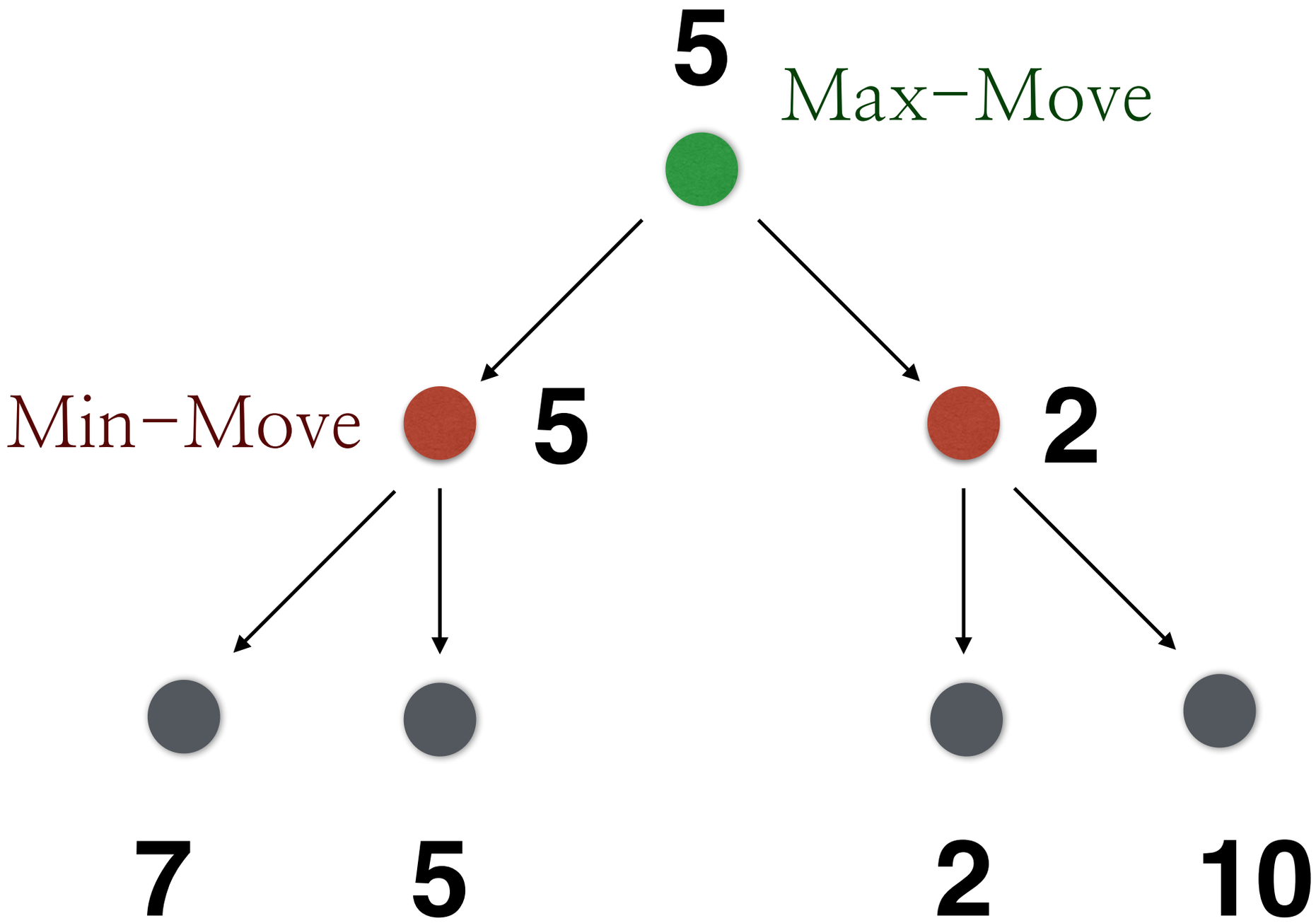
The optimal outcome that the max player can achieve by going into the subtree of that node, assuming that each player is perfectly rational.

C wants min, R wants max, C goes first

	C1	C2
R1	7	5
R2	2	10

Moves: (C1, R1)


Max-Move
Min-Move
7
5
2
10


5
Max-Move
Min-Move
5
2
7
5
2
10

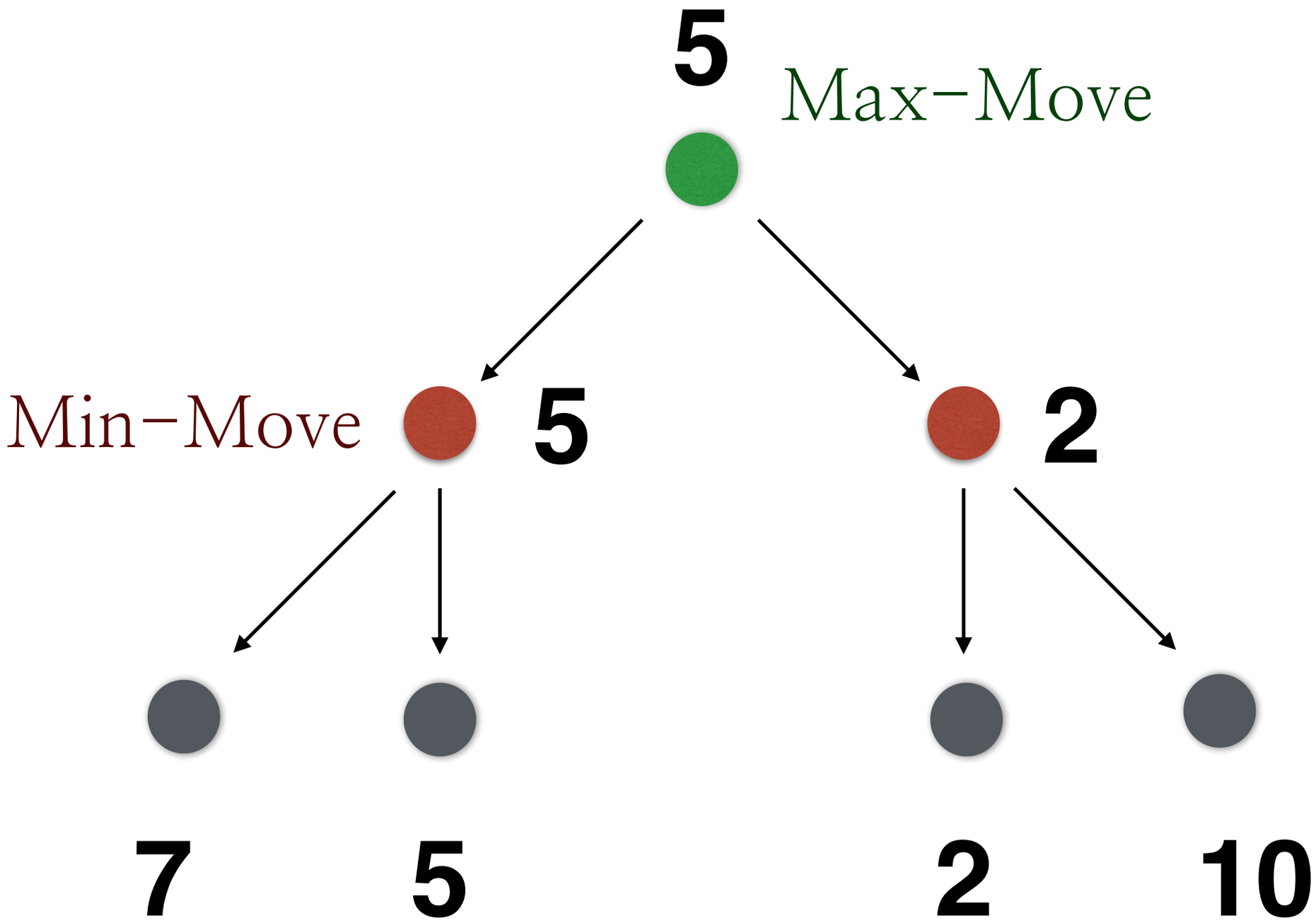
Minimax Values

Minimax(node):

- If node is terminal, then Payoff(node)
- If node is controlled by the Max player, then the maximum of all Minimax(sub-node)
- If node is controlled by the Min player, then the minimum of all Minimax(sub-node)

Minimax Algorithm

```
def minimax(node):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, minimax(n))
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, minimax(n))
        return value
    else:
        error
```


5
Max-Move
Min-Move
5
2
7
5
2
10

Minimax Algorithm

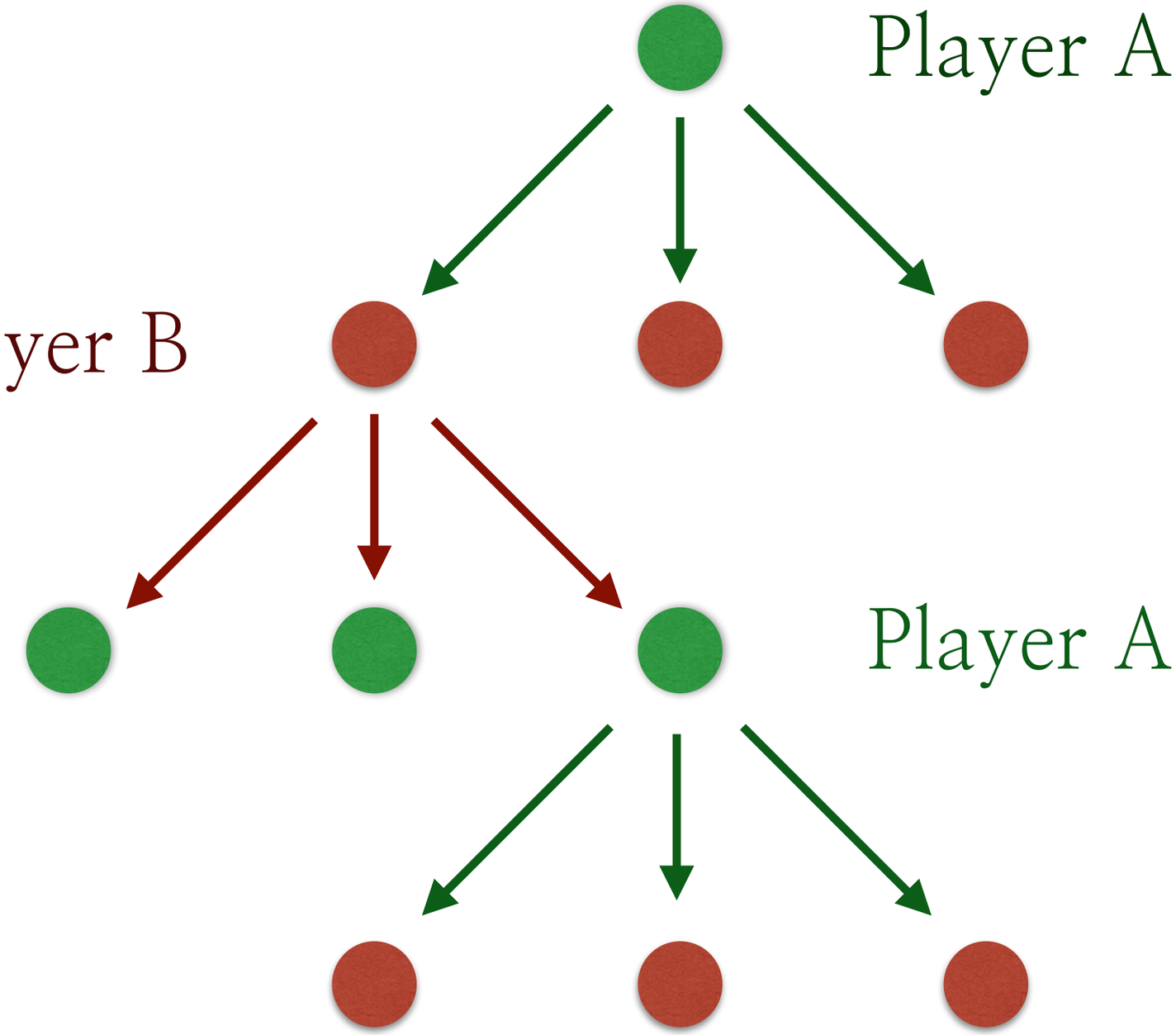
Remember:

We compute values for all these trees just to make one decision: what do we do (as the max player) at the root?

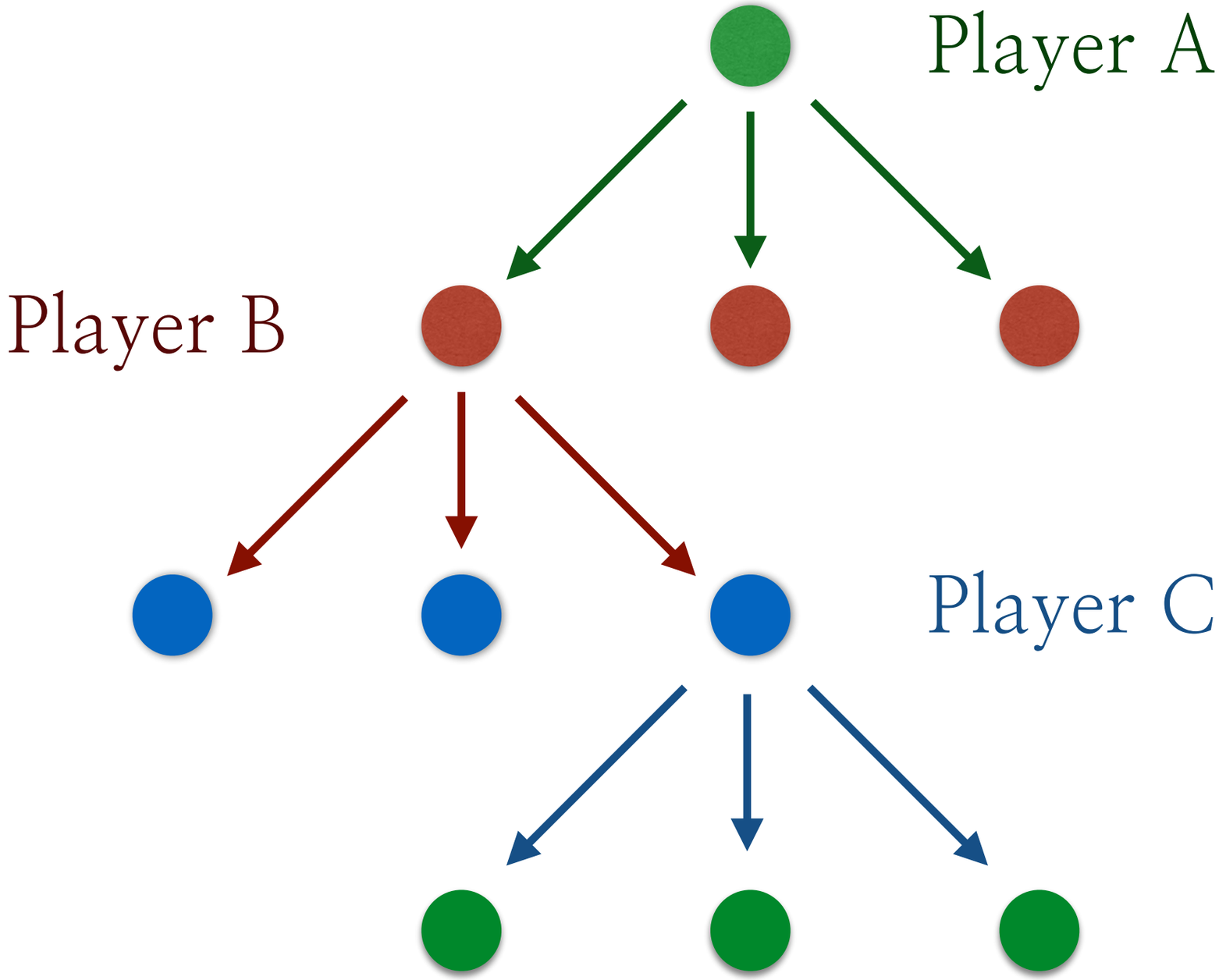
Minimax Algorithm

- DFS? BFS?
- Complexity?
- Tree expansions are just simulations
- You don’t need to understand the game!

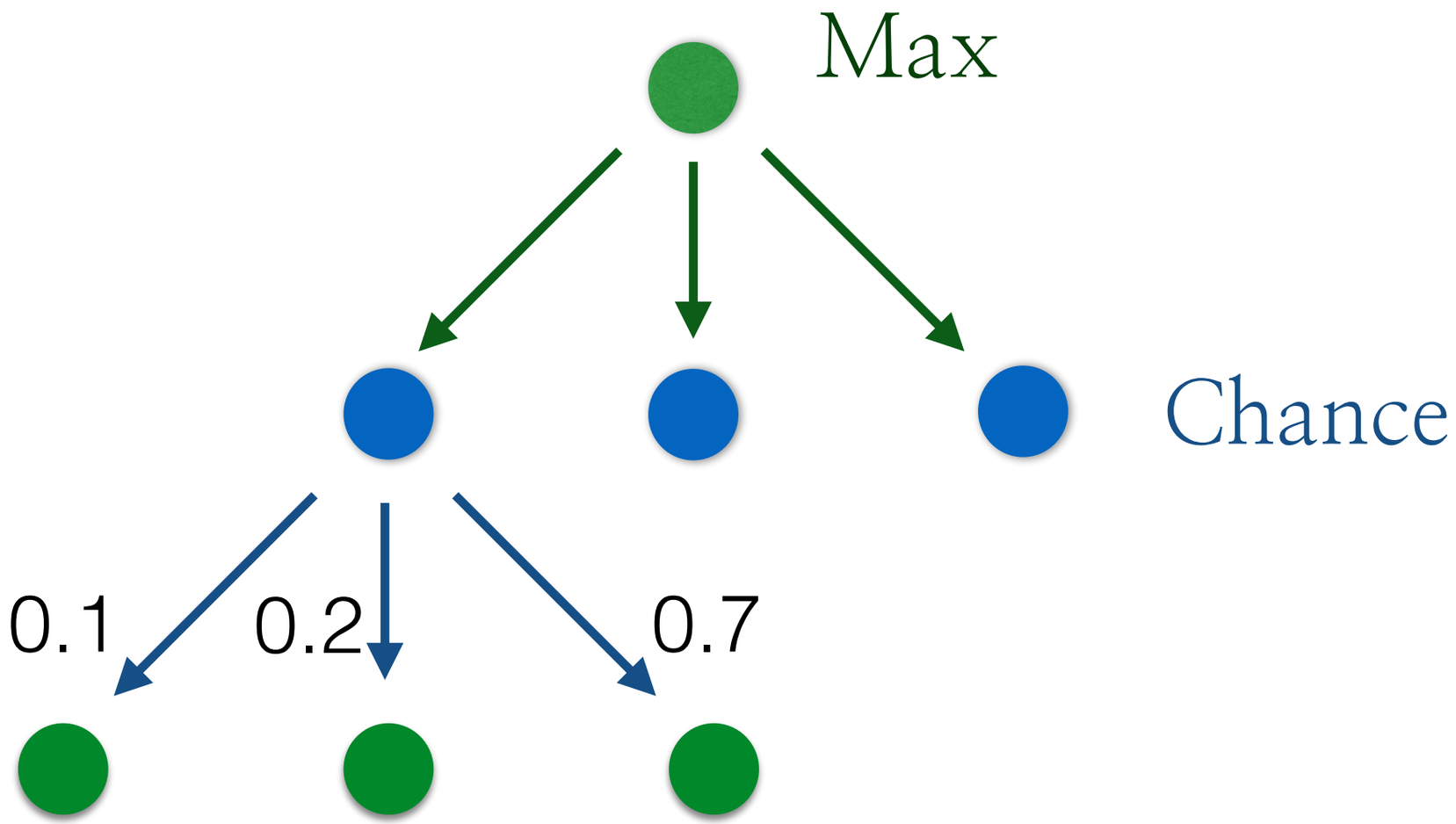
Extension I: Non Zero-Sum



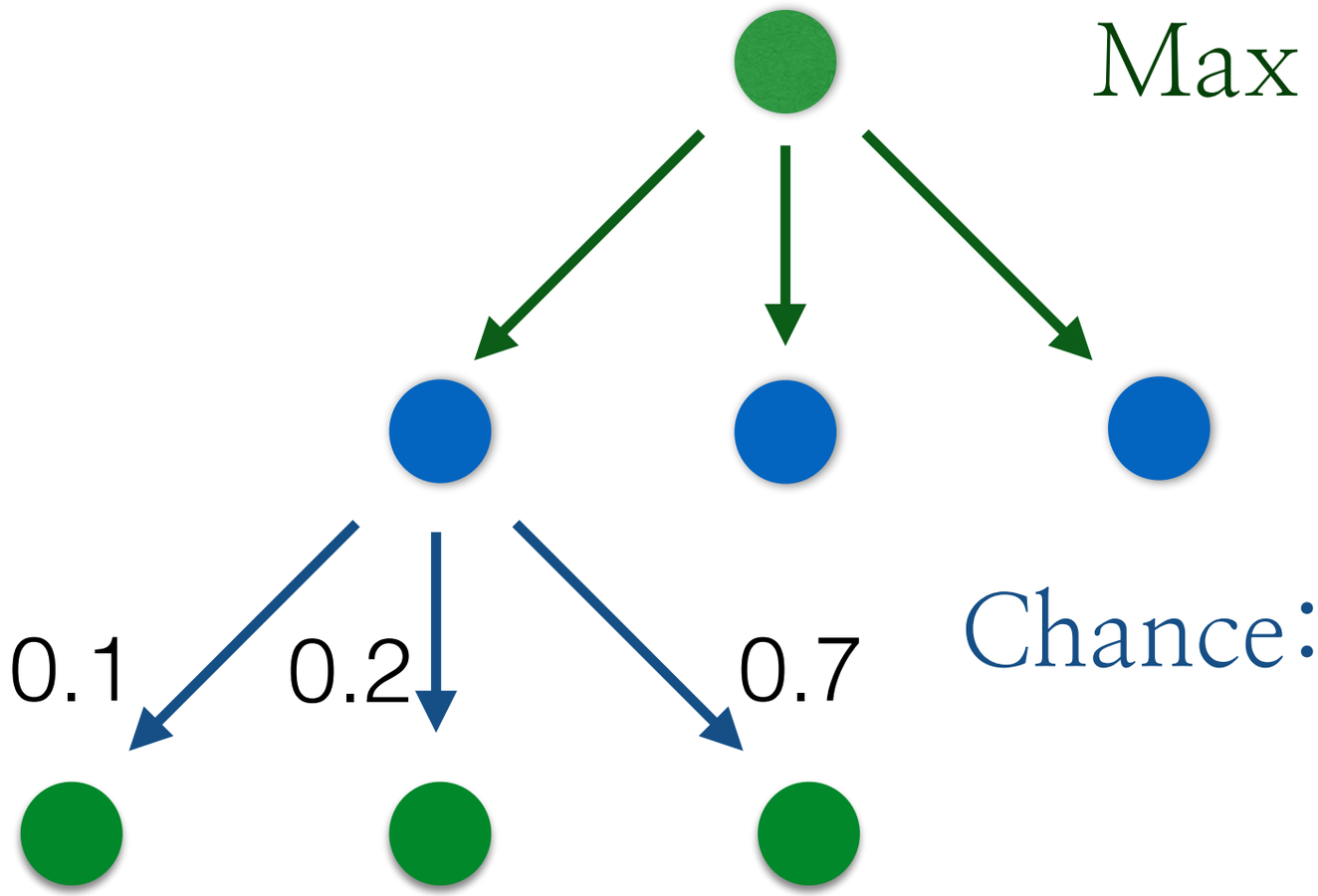
Extension II: Multiple Players



Extension III: Stochastic Games



Expectimax



Chance: use expectation

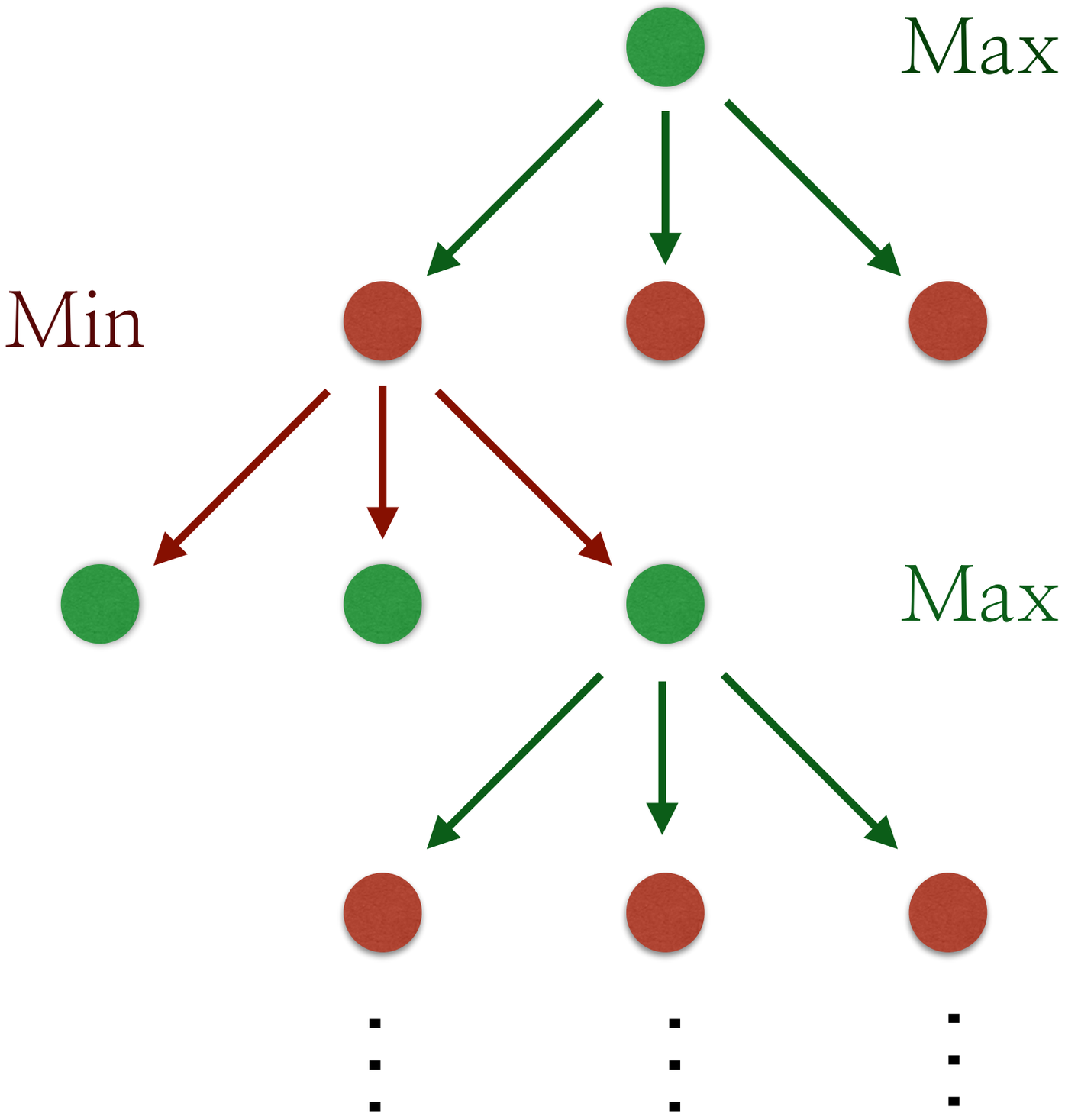
Expectimax Algorithm

```
def expectimax(node):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, expectimax(n))
        return value
    elif chance_player(node):
        value = 0
        for n in children(node):
            value = value + expectimax(n)*chance(n)
        return value
    else:
        error
```

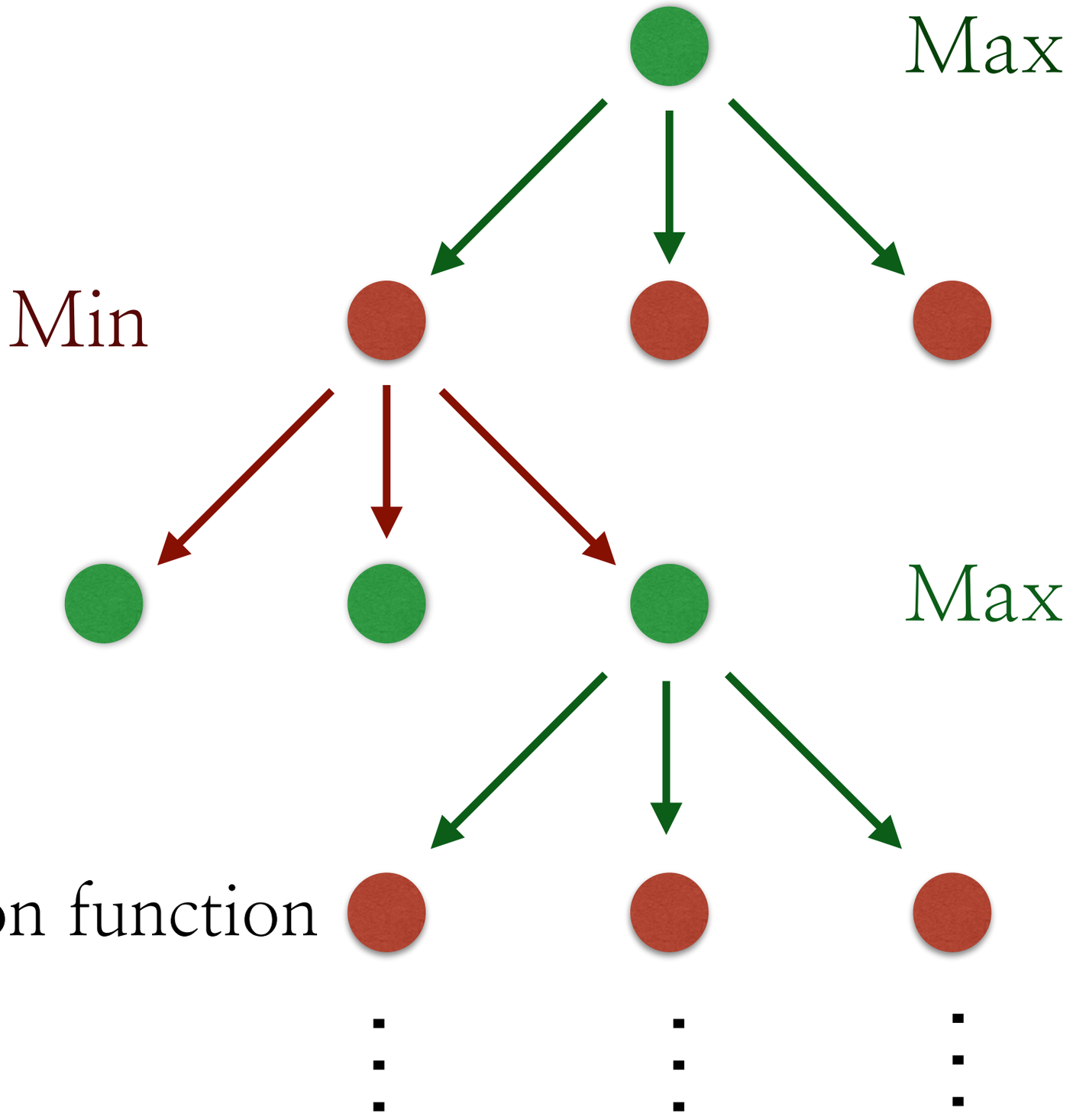
Trees are too large!

- Vertical Pruning: Cut-off at some fixed depth and use heuristics to evaluate the intermediate nodes.
- Horizontal Pruning: Alpha-Beta Pruning

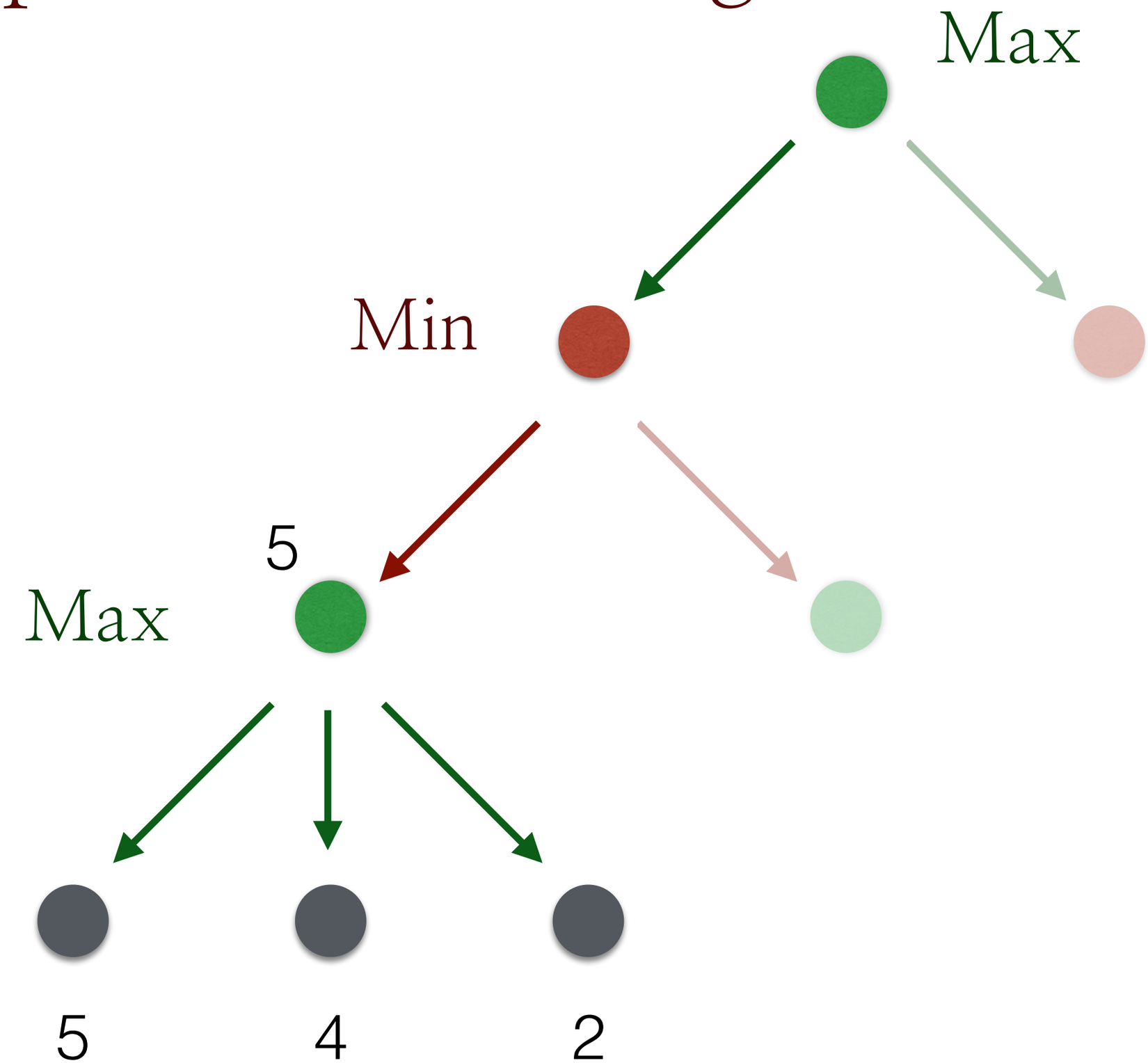
Vertical Cut-off



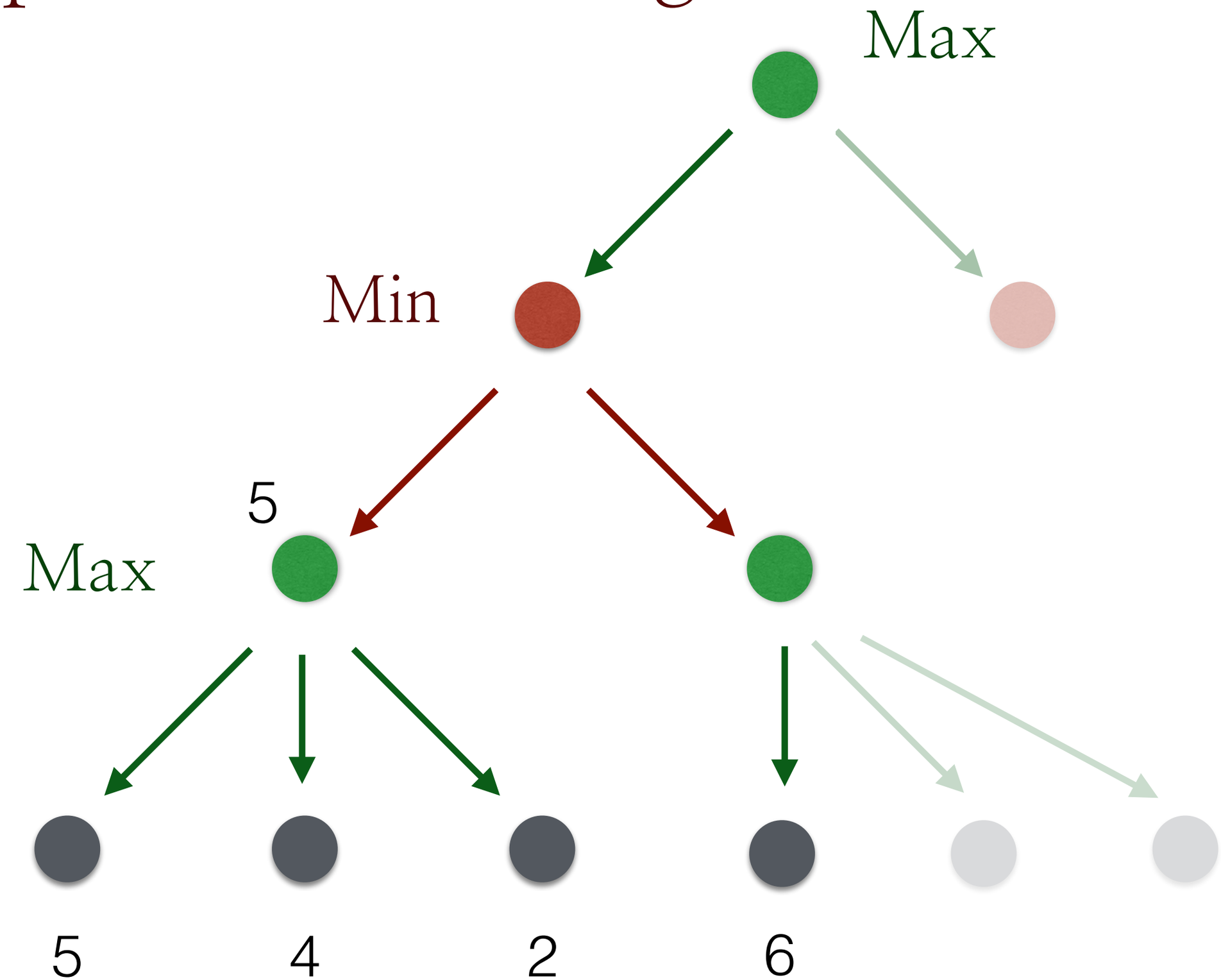
Vertical Cut-off



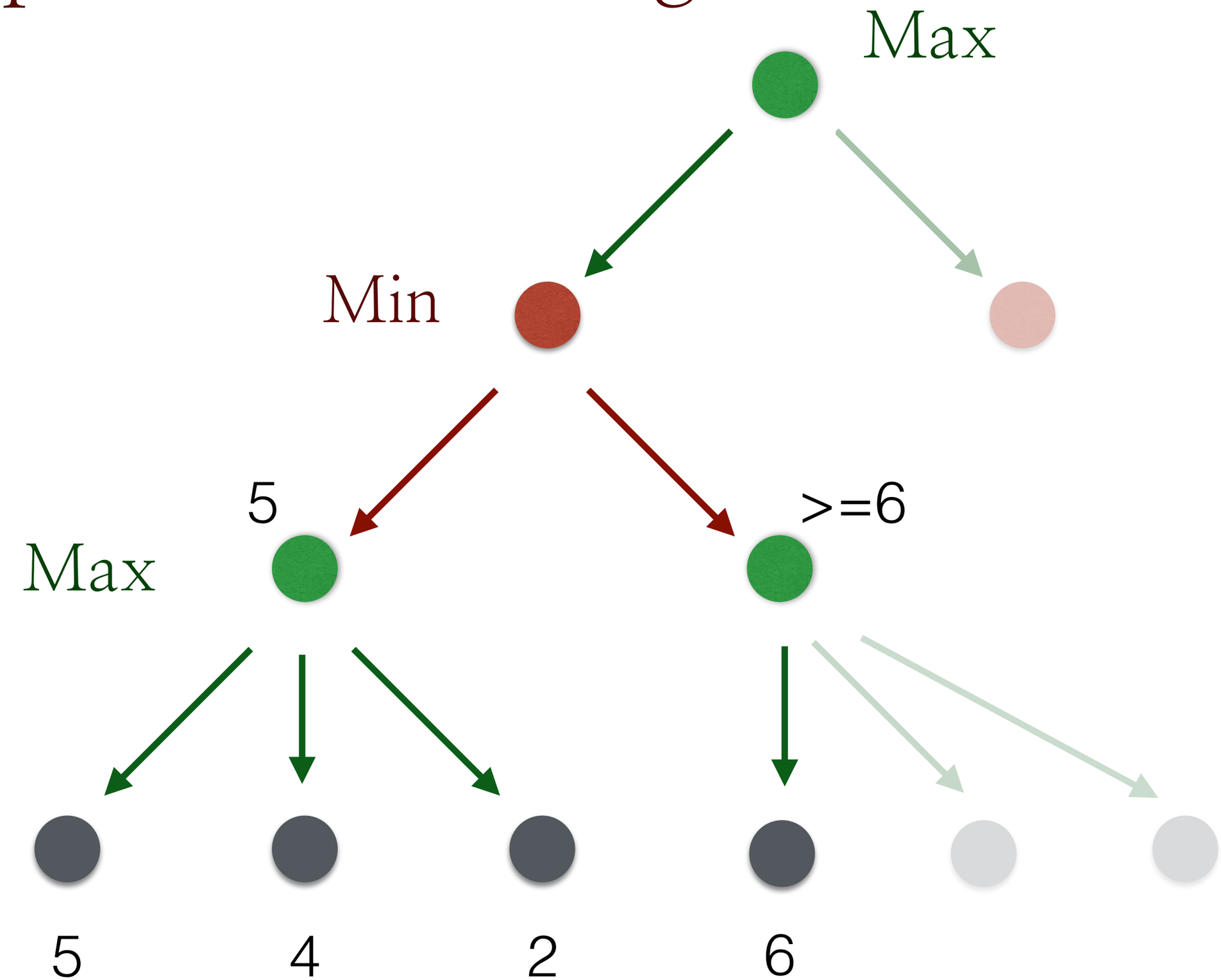
Alpha-Beta Pruning



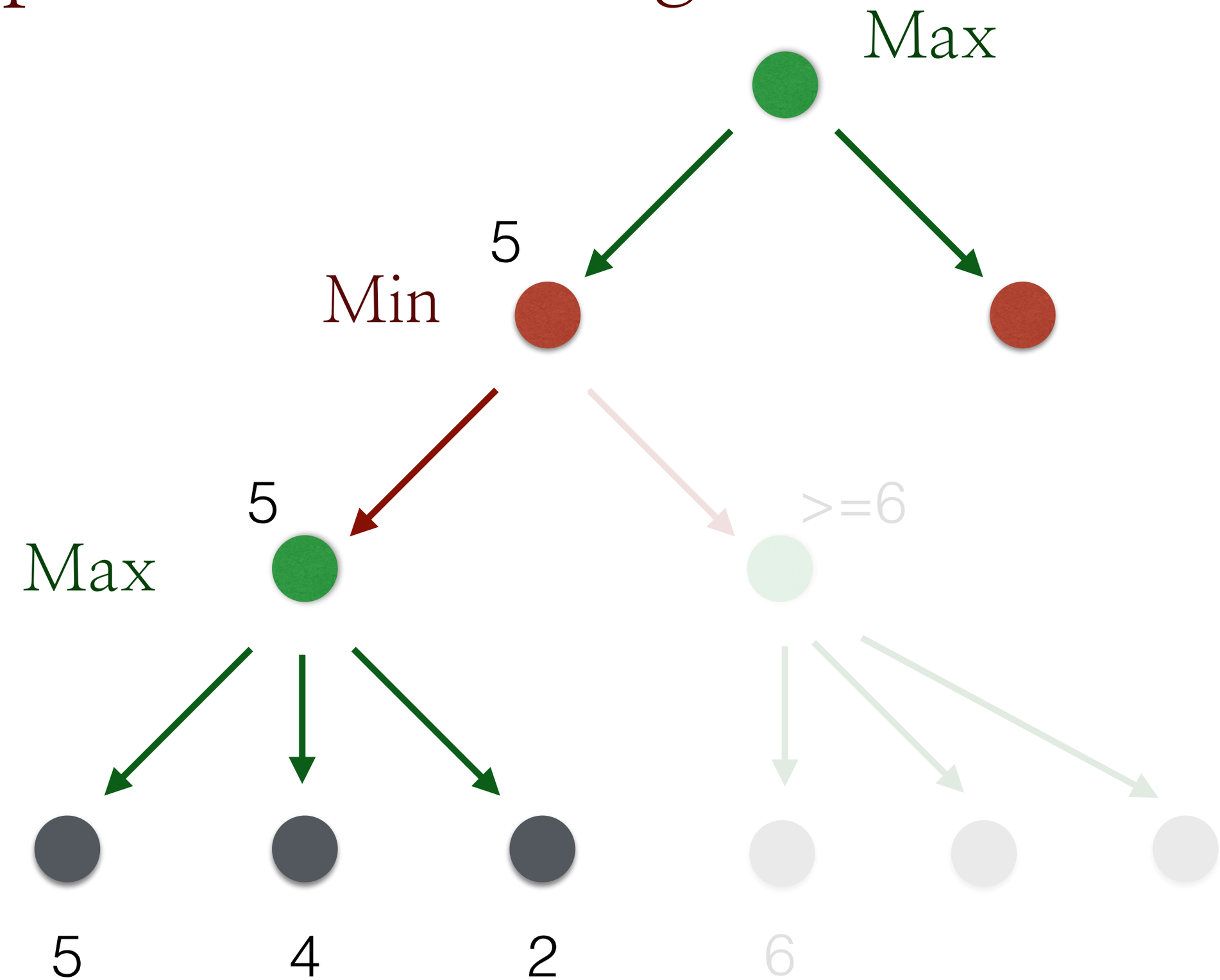
Alpha-Beta Pruning



Alpha-Beta Pruning



Alpha-Beta Pruning



```
alpha = -infinity '''fallback for Max'''
beta = infinity '''fallback for Min'''
node = root

def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```


. e4 c6 2. d4 d5 3. Nc3 dxe4