TDDC17 LE4 HT2024 – Search II

Fredrik Heintz

Dept. of Computer Science
Linköping University

fredrik.heintz@liu.se
@FredrikHeintz

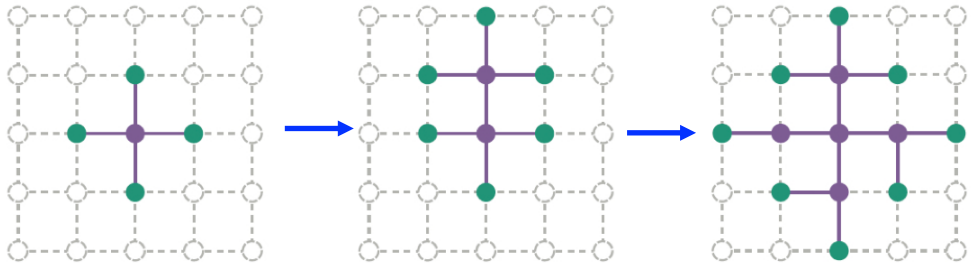
Outline:

- **Informed Search Algorithms (Ch 3)**
- **Search in Complex Environments (Ch 4)**
- **Adversarial Search and Games (Ch 6)**

LINKÖPING UNIVERSITY

Intuitions behind Heuristic Search

Separation property of graph search



Systematic Search
through the state space

Find a heuristic measure *h(n)* which estimates how close a node *n* in the frontier is to the nearest goal state and then order the frontier queue accordingly relative to closeness.

The evaluation function *f(n)*, previously discussed will include *h(n):*

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

h(n) is intended to provide domain specific hints about location of goals

Recall Best-First Search

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Evaluation function: f(n)

Minimum of f(n) first

Different evaluation functions f(n), will generate different algorithms

Heuristic Search Algorithm:

f(n) = ... + h(n)

Greedy Best-First Search

```
function GREEDY-BEST-FIRST-SEARCH (problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Evaluation function: f(n)

Minimum of f(n) first

Greedy Best-First Search:

$$f(n) = h(n)$$

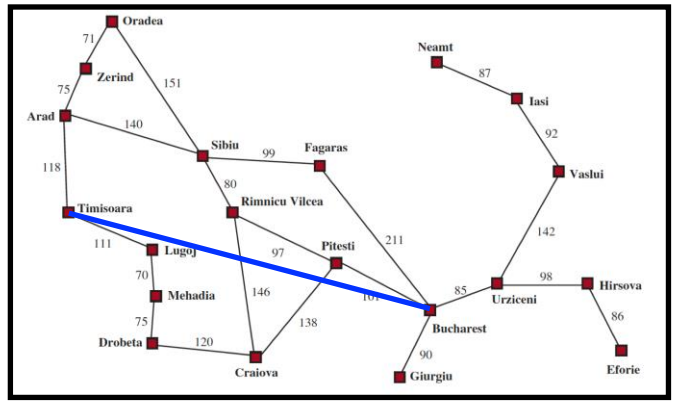
Don't care about anything except how close a node is to a goal!

Romania Travel Problem

Let's find a heuristic!

Straight line distance from city *n* to goal city *n'*

Assume the cost to get somewhere is a function of the distance traveled



Straight line distance to Bucharest from any city

$h_{SLD}()$

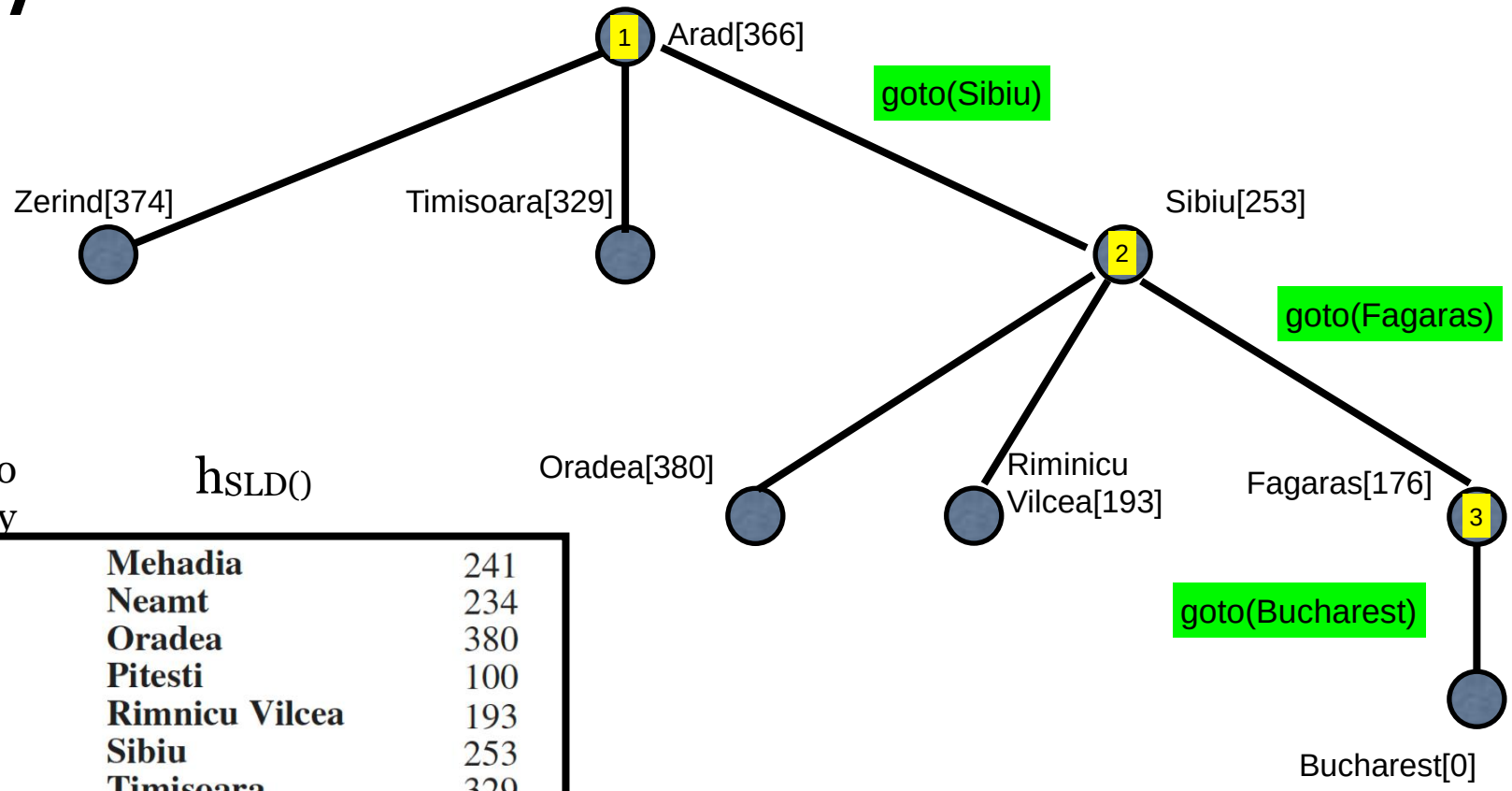
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Heuristic:

$f(n) = h_{SLD}(n)$

Notice the SLD under estimates the actual cost!

Greedy Best-First Search: Romania

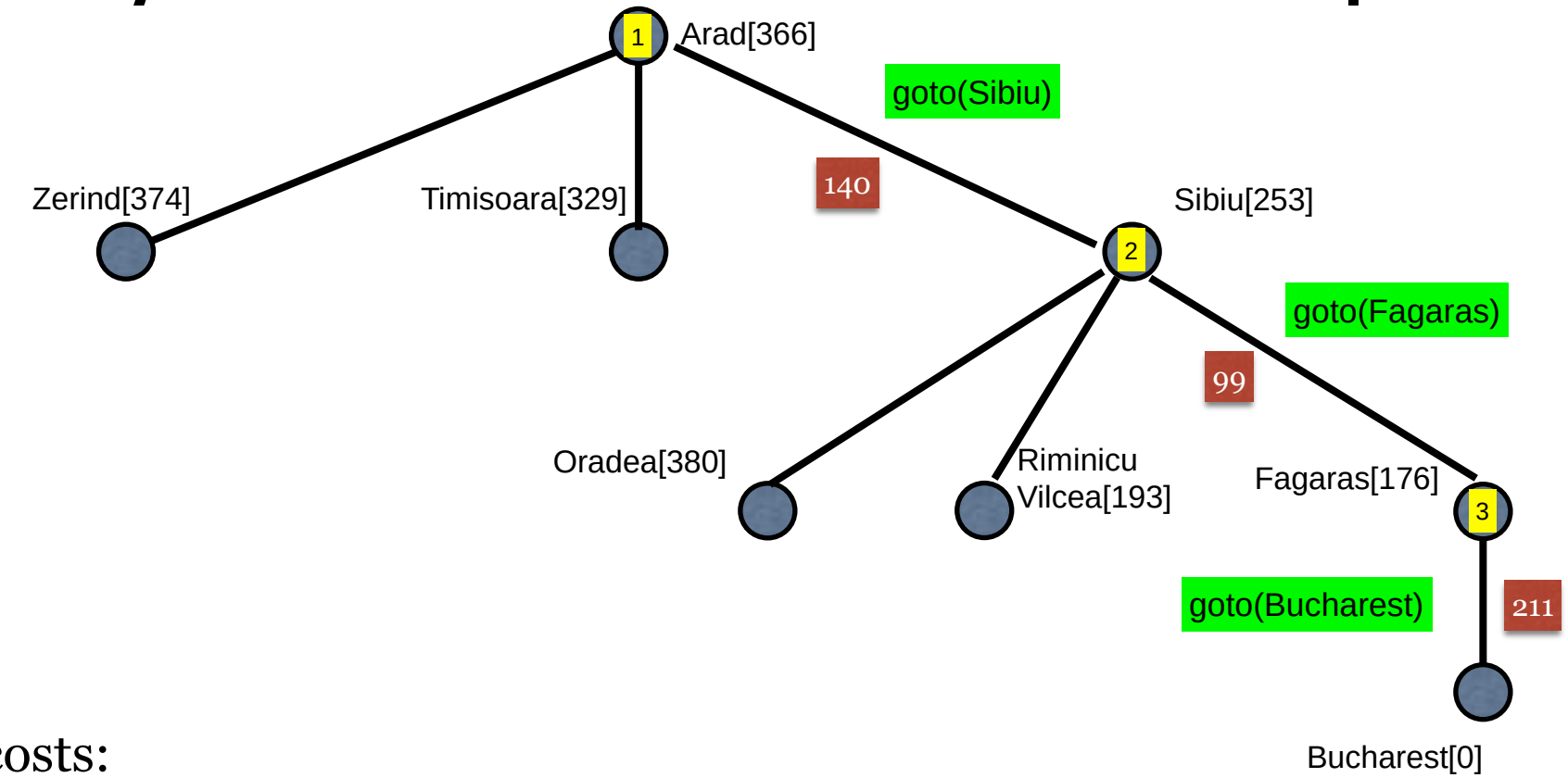


Straight line distance to Bucharest from any city

$h_{SLD}()$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Is Greedy Best-First Search Cost-Optimal?



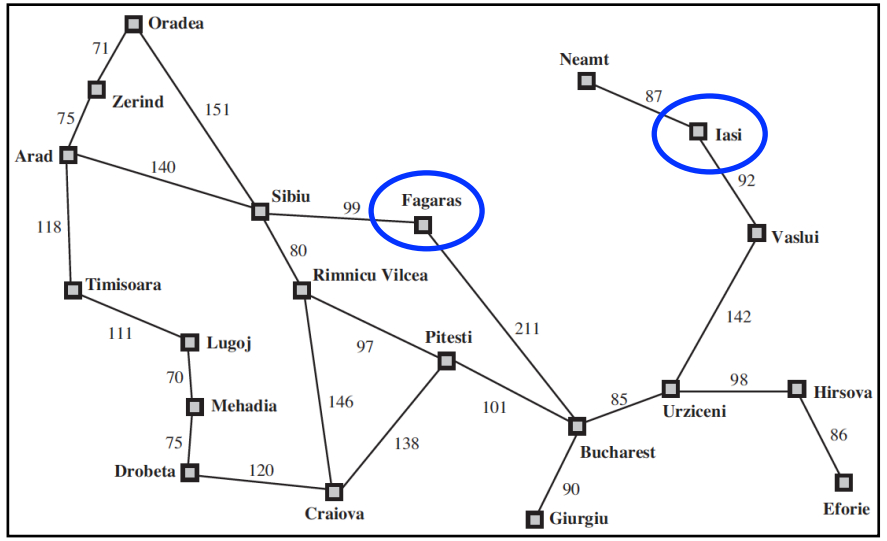
No, the actual costs:
Path Chosen: Arad-Sibiu-Fagaras-Bucharest = **450**
Optimal Path: Arad-Sibiu-Rimnicu Vilcea-Pitesti-Bucharest = **418**

The search cost is minimal but not optimal! What's missing?

Is Greedy Best-First Search Complete?

- GBF (Graph search) is complete in finite spaces but not in infinite spaces
- GBF (Tree-like search) is not even complete in finite spaces.

Consider going from Iasi to Fagaras?



Neamt is chosen 1st because *h(Neamt)* is closer than *h(Vaslui)*, but Neamt is a dead-end. Expanding Neamt still puts Iasi 1st on the frontier again since *h(Iasi)* is closer than *h(Vaslui)*...which puts Neamt 1st again!

- GBF (Graph Search): Time/Space Complexity: $O(|V|)$
- GBF (Tree-Like Search): Time/Space Complexity: $O(b^m)$
- With good heuristics, complexity can be reduced substantially

* m -maximum length of any path in the search space (possibly infinite)

Improving Greedy Best-First Search

Greedy Best-First Search finds a goal as fast as possible by using the *h(n)* function to estimate *n*'s closeness to the goal.

Greedy Best-First Search chooses any goal node without concerning itself with the shallowness of the goal node or the cost of getting to *n* in the 1st place.

Rather than choosing a node based just on distance to the goal we could include a quality notion such as *expected depth of the nearest goal*

g(n) - the actual cost of getting to node *n*
h(n) - the estimated cost of getting from *n* to a goal state

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

f(n) is the estimated cost of the cheapest solution through *n*

A* (Graph)Search

```
function A*-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Evaluation function: f(n)

Minimum of f(n) first

Note: This algorithm only works as is, if the heuristic function *h(n)* is consistent. More on this soon.

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

g(n)

A*-1

A* Tree-like Search

(a) The initial state

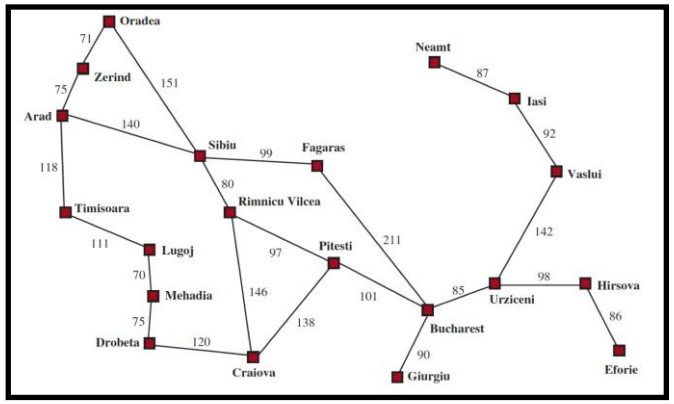
366=0+366

g(Arad)+h(Arad)

Heuristic (with Bucharest as goal):

$f(n) = g(n) + h(n)$
$g(n)$ - Actual distance from root node to n
$h(n)$ - $h_{SLD}(n)$ straight line distance from n to Bucharest

g(n)



Straight line distance to Bucharest from any city

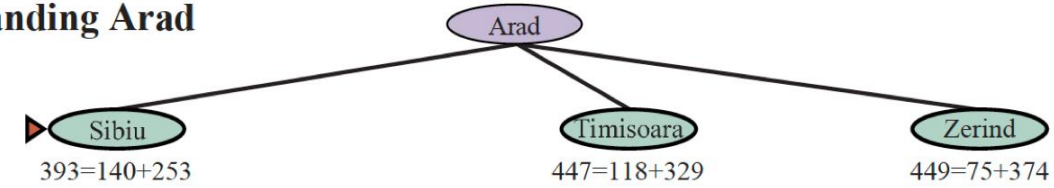
h(n) = $h_{SLD}(n)$

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

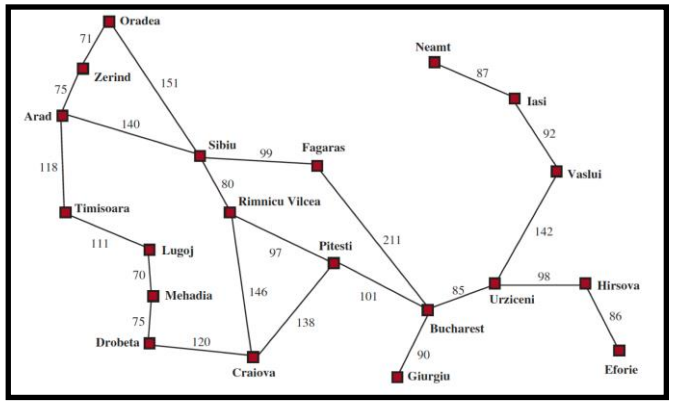
A*-2

A* Tree-like Search

(b) After expanding Arad



g(n)



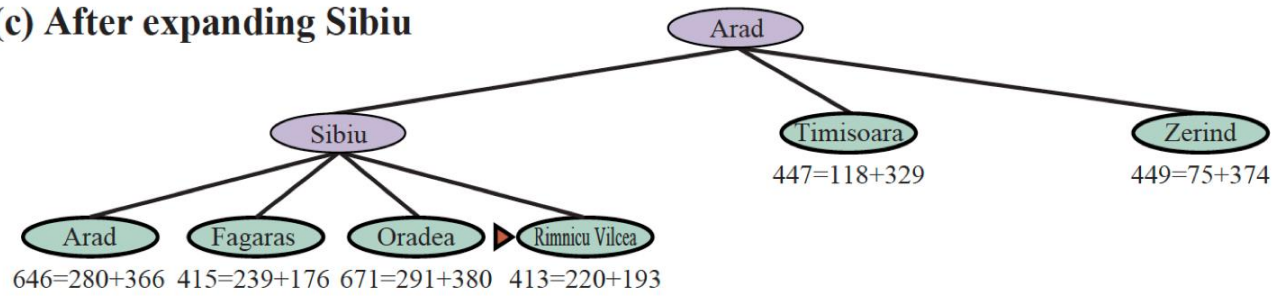
Straight line distance to Bucharest from any city

h(n) = $h_{SLD}(n)$

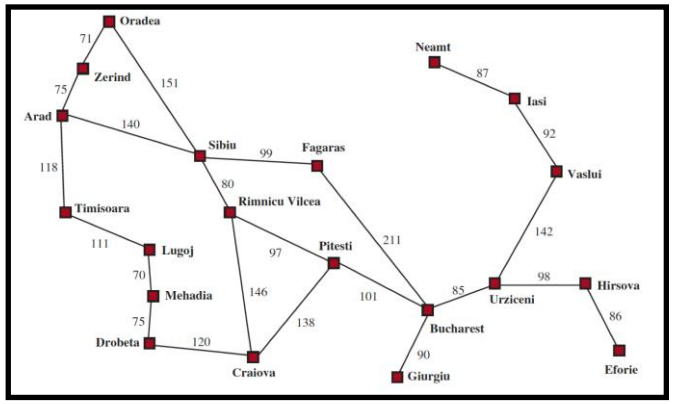
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A*-3

A* Tree-like Search



g(n)

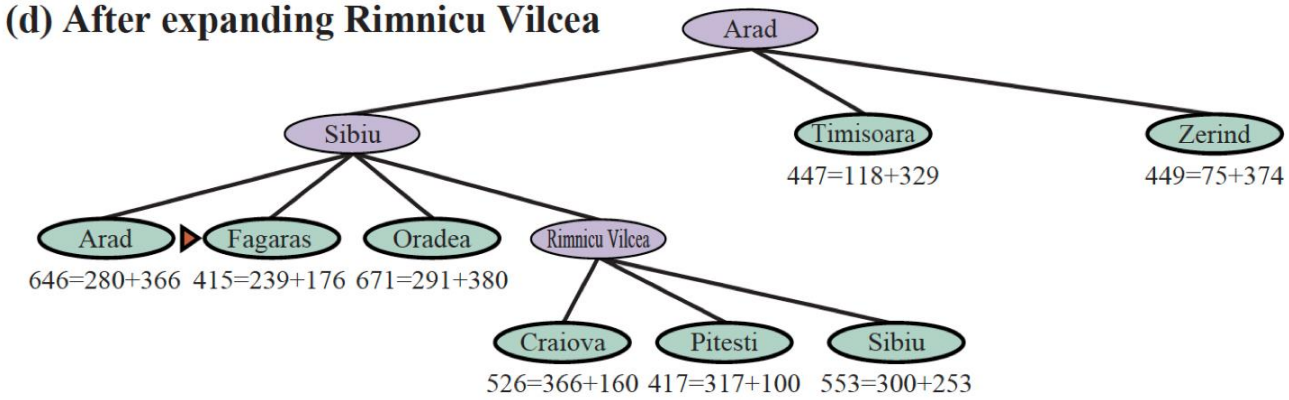


Straight line distance to Bucharest from any city

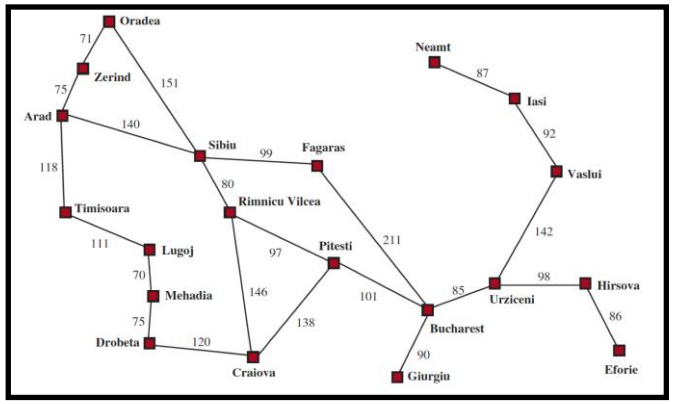
h(n) = $h_{SLD}(n)$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A*-4



g(n)



Straight line distance to Bucharest from any city

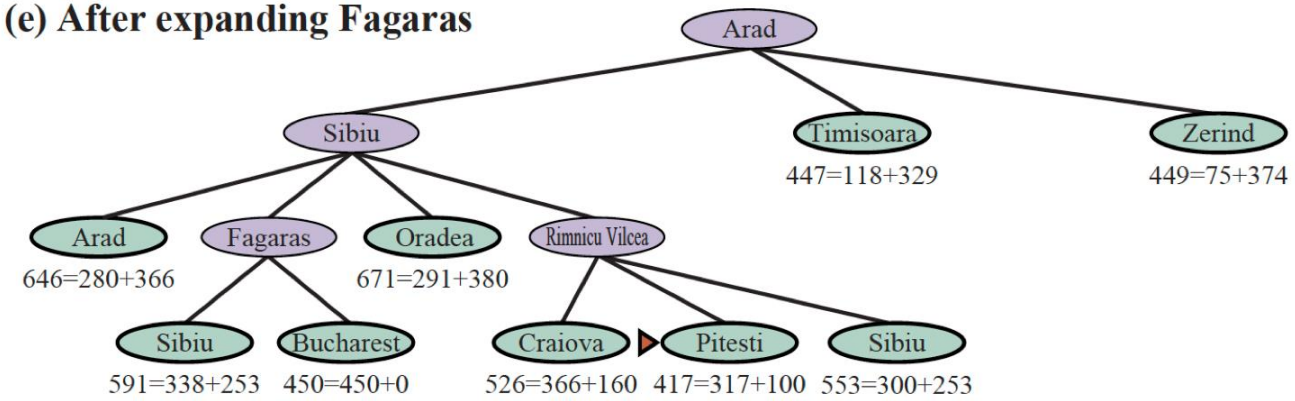
h(n) = $h_{SLD}(n)$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

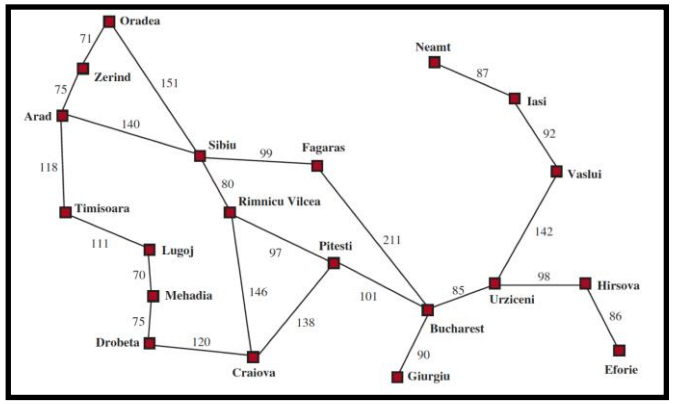
A*-4

A* Tree-like Search

(e) After expanding Fagaras



g(n)



Straight line distance to Bucharest from any city

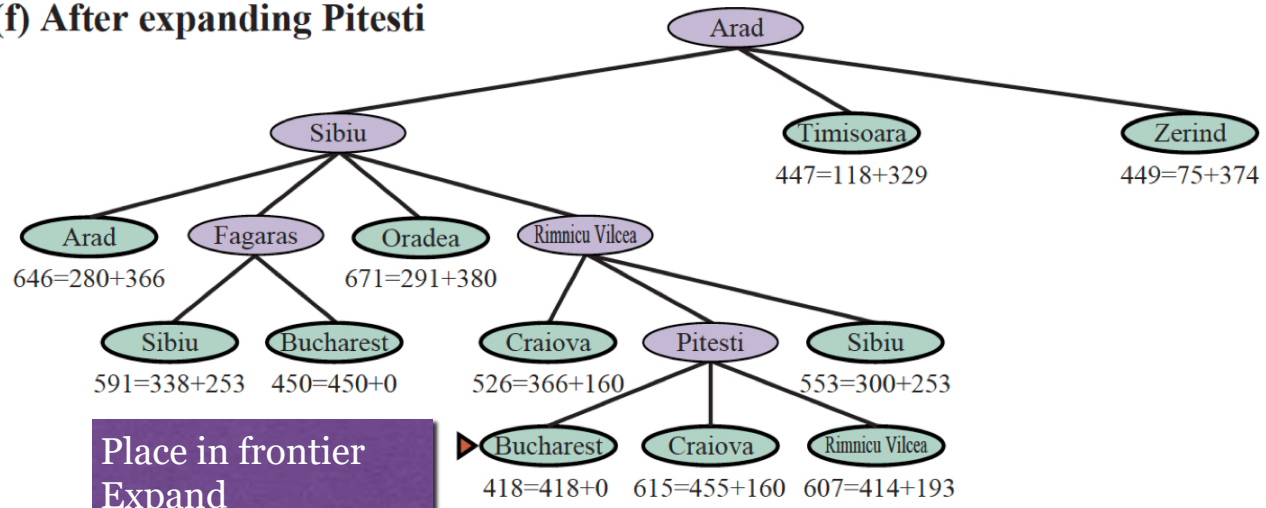
h(n) = $h_{SLD}(n)$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A*-6

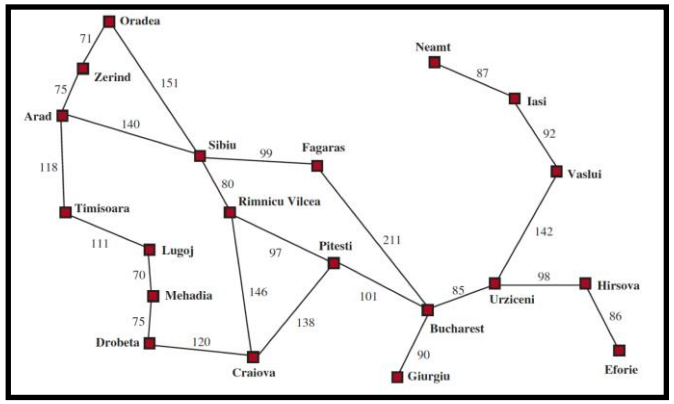
A* Tree-like Search

(f) After expanding Pitesti



Place in frontier
Expand
Late Testing Goal

g(n)



Straight line distance to Bucharest from any city

h(n) = $h_{SLD}(n)$

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Admissibility

An *admissible heuristic* is one that never overestimates the cost to reach a goal (It is optimistic)

h(n) takes a node *n* and returns a non-negative real number that is an *estimate* of the cost of the least-cost path from node *n* to a goal node

h(n) is an *admissible heuristic*, if *h(n)* is always less than or equal to *the actual* cost of a least-cost path from node *n* to a goal.

Admissibility does not ensure that every intermediate node selected from the frontier is on an optimal path from the start node to the goal node. It may change its mind about which partial path is best while searching and the frontier may include multiple paths to the same state.

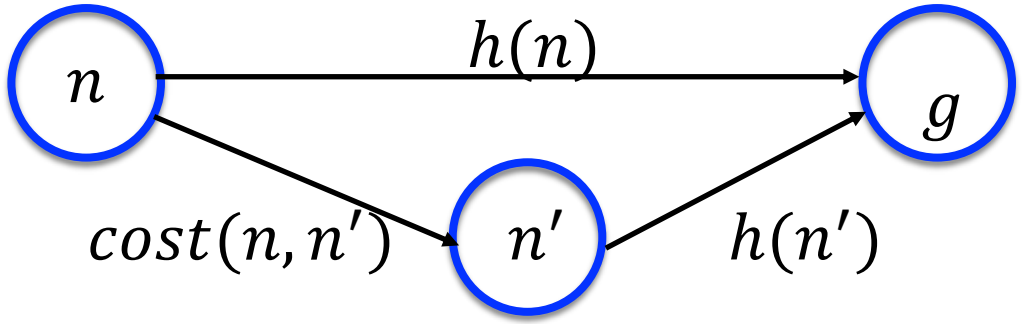
This implies that the A* (graph) search algorithm may not be cost-optimal for A* with $f(n) = g(n) + h(n)$, where $h(n)$ is only admissible. Additional bookkeeping is required. A*(tree-like) search is cost-optimal, but is less efficient.

Admissibility with A*(tree-like) search does ensure that first solution found will be cost-optimal

Consistency

A *consistent heuristic* is a non-negative function $h(n)$ on a node n that satisfies the constraint: $h(n) \leq cost(n, n') + h(n')$ for any two nodes n and n', where $cost(n, n')$ is the cost of the least-cost path from n to n'.

The estimated cost of going from n to a goal should not be more than the estimated cost of first going to n' and then to a goal



Triangle inequality

Consistency/ Monotonicity

Consistency is guaranteed if the heuristic function satisfies the *monotone restriction*: $h(n) \leq c(n, a, n') + h(n'), \forall a, n, n'$

Easier to check than consistency: Just check arcs in state space graph rather than all pairs of states.

If $h(n)$ is a consistent heuristic then it is also an admissible heuristic

Consistency/Monotonicity guarantees:

- *f*-paths selected from the frontier are monotonically non-decreasing (*f*-values do not get smaller)
- First time we reach a state on the frontier it will be on an optimal path, so
  - We never need to re-add a state to the frontier
  - We never need to change an entry in reached

This implies that the A* (graph) search algorithm can be used for A* with $f(n) = g(n) + h(n)$, where $h(n)$ is consistent.

A* Proof of Optimality (Tree-like Search)

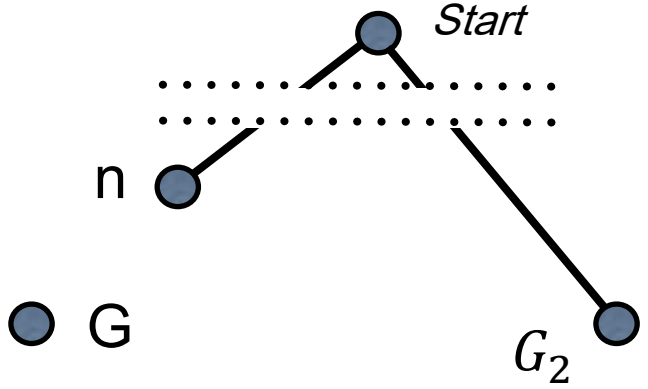
A* using (Tree-Like) SEARCH is cost optimal if h(n) is admissible

Proof:

Assume the cost of the optimal solution is C*.
Suppose a suboptimal goal node G_2 appears on the fringe.

Since G_2 is suboptimal and h(G_2)=0 (G_2 is a goal node),
$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$
Now consider the fringe node n that is on an optimal solution path. If h(n) does not over-estimate the cost of completing the solution path then $f(n) = g(n) + h(n) \leq C^*$

Then $f(n) \leq C^* \leq f(G_2)$
So, G_2 will not be expanded and A* is optimal!



See example:
n = Pitesti (417)
G_2 = Bucharest (450)

A* Proof of Optimality (Graph Search)

A* using GRAPH-SEARCH is cost- optimal if $h(n)$ is consistent (monotonic)

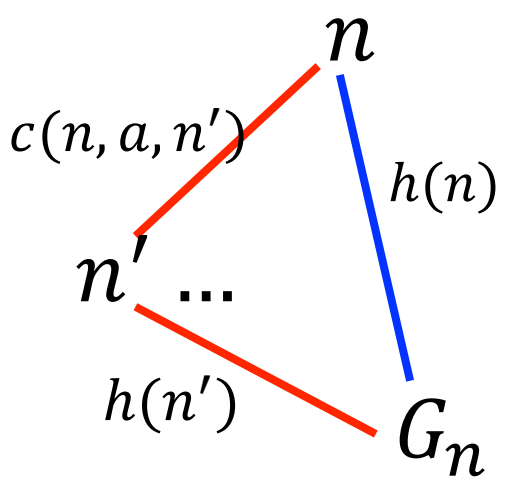
Step Cost

$$h(n) \text{ is consistent } h(n) \leq c(n, a, n') + h(n'), \forall a, n, n'$$

Step cost:

successors(n):

Triangle inequality argument:
Length of a side of a triangle is always less than the sum of the other two.



As one extends a path from n to n'
this assures that $f(n) \leq f(n')$:
$g(n) + h(n) \leq g(n) + c(n, a, n') + h(n')$

... n_k

:Goal node closest to n

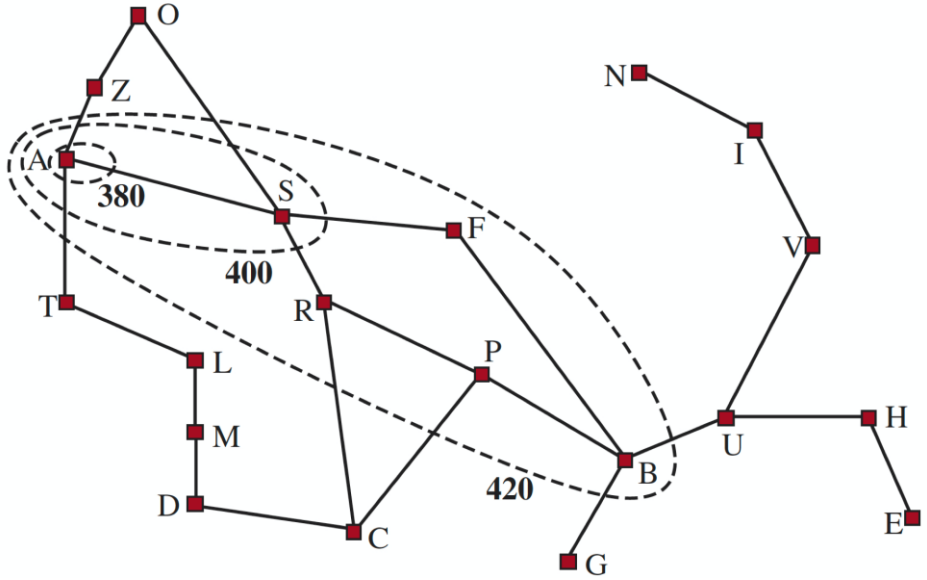
Estimated cost of getting to G_n from n: $f(n)$, can not be more than going through a successor of n to G_n: $f(n')$, otherwise it would violate the property that $h(n)$ is a lower bound on the cost to reach G_n

Optimality of A* (Graph Search)

Steps to show in the proof:

- If *h(n)* is consistent, then the values *f(n)* along any path are non-decreasing
- Whenever A* selects a node *n* for expansion from the frontier, the optimal path to that node has been found

If this is the case, the values along any path are non-decreasing and A* fans out in concentric bands of increasing f-cost



Map of Romania showing contours at f=380, f=400, and f=420 with Arad as start state. Nodes inside a given contour have f-costs $\leq$ to the contour value.

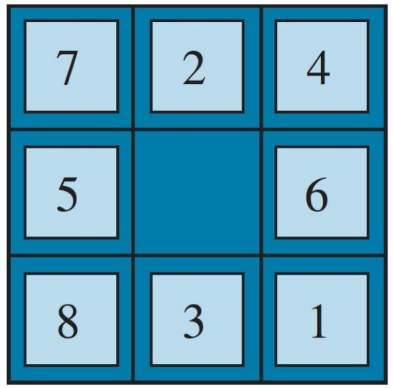
Some properties of A*

- Cost-Optimal -
  - for a given admissible heuristic (tree-like search)
  - for a given consistent heuristic (tree-like, graph-search)
  - Consistent heuristics are admissible heuristics but not vice-versa.
- Complete - Eventually reach a contour equal to the path of the least-cost to the goal state.
- Optimally efficient - No other algorithm, that extends search paths from a root is guaranteed to expand fewer nodes than A* for a given heuristic function.
- The exponential growth for most practical heuristics will eventually overtake the computer (run out of memory)
  - The number of states within the goal contour is still exponential in the length of the solution.
  - There are variations of A* that bound memory….

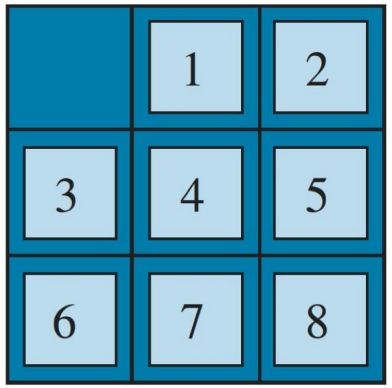
Finding Admissible Heuristics

h(n) is an admissible heuristic if it never over-estimates the cost to reach the goal from *n*.

Admissible Heuristics are optimistic because they always think the cost of solving a problem is less than it actually is.



Start State

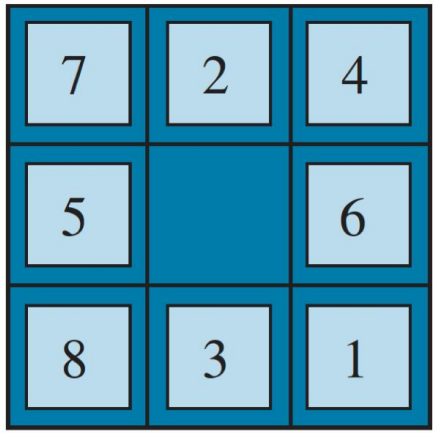


Goal State

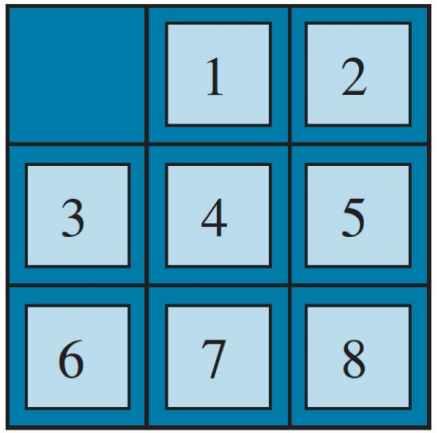
The 8 Puzzle

How would we choose an admissible heuristic for this problem?

8-Puzzle Heuristics



Start State



Goal State

True solution is 26 moves. (C*)

$h_1(n)$: The number of pieces that are out of place.
(8) Any tile that is out of place must be moved at least once. Definite under estimate of moves!

$h_2(n)$: The sum of the Manhattan distances for each tile that is out of place.
(3+1+2+2+2+3+3+2=18) . The Manhattan distance is an under-estimate because there are tiles in the way.

Inventing Admissible heuristics: Problem relaxation

- A problem with fewer restrictions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is in fact an admissible heuristic to the original problem

If the problem definition can be written down in a formal language, there are possibilities for automatically generating relaxed problems automatically!

Sample rule:
A tile can move from square A to square B if
A is horizontally or vertically adjacent to B
and B is blank

Some Relaxations

Sample rule:

A tile can move from square A to square B if
A is horizontally or vertically adjacent to B
and B is blank

---

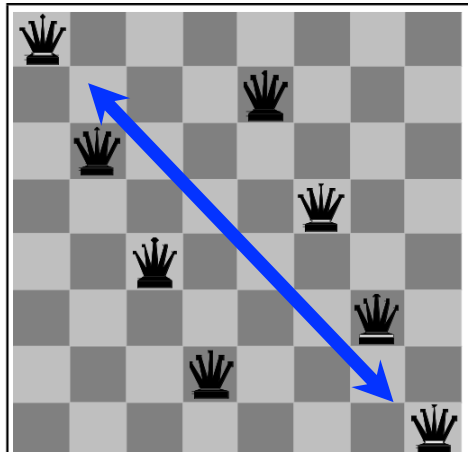
1. A tile can move from square A to square B if A is adjacent to B
2. A tile can move from square A to square B if B is blank
3. A tile can move from square A to square B

(1) gives us Manhattan distance: $h_2(n)$
(3) gives us misplaced tiles: $h_1(n)$

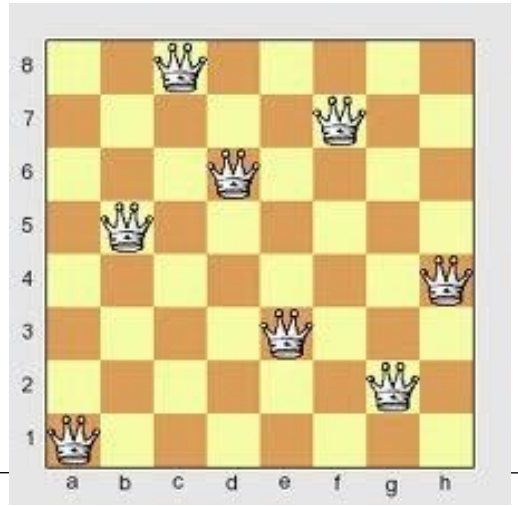
Search in Complex Environments

Chapter 4

Local Search: 8 Queens Problem



Bad Solution



Good Solution

Problem:
Place 8 queens on a chessboard such that
No queens attacks another

- Local Search:
  - the path to the goal is irrelevant!
  - we do not care about reached states
  - complete state formulation is a straightforward representation:
    - 8 queens, one in each column
  - operate by searching from start state to neighbouring states, choose the best neighbour so far, repeat

8 Queens is a candidate for use of local search!

8^8 (about 16 million configurations)

Local Search Techniques

- Advantages:
  - They use very little memory
  - Often find solutions in large/infinite search spaces where systematic algorithms would be unreasonable
  - Can be used to solve optimisation problems
- Disadvantages
  - Since they are not systematic they may not find solutions because they leave parts of the search space unexplored.
  - Performance is dependent on the topology of the search space
  - Search may get stuck in local optima

Global Optimum: The best possible solution to a problem.

Local Optimum: A solution to a problem that is better than all other solutions that are slightly different, but worse than the global optimum

Greedy Local Search: A search algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems. (They may also get stuck!)

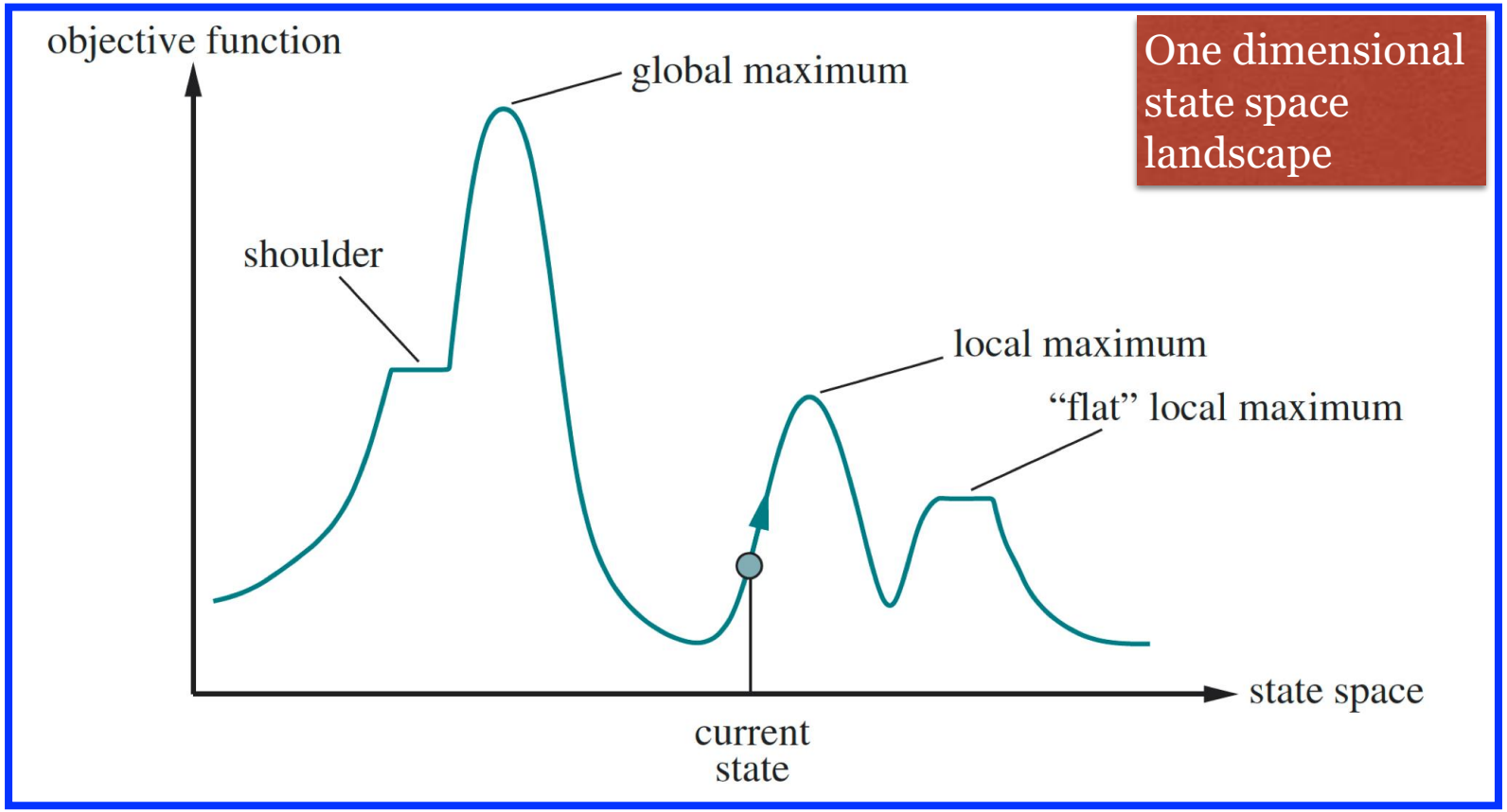
Hill-Climbing Algorithm (steepest ascent version)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  current ← *problem*.INITIAL
  while *true* **do**
    neighbor ← a highest-valued successor state of *current*
    if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
    current ← *neighbor*

When using heuristic functions: steepest descent version

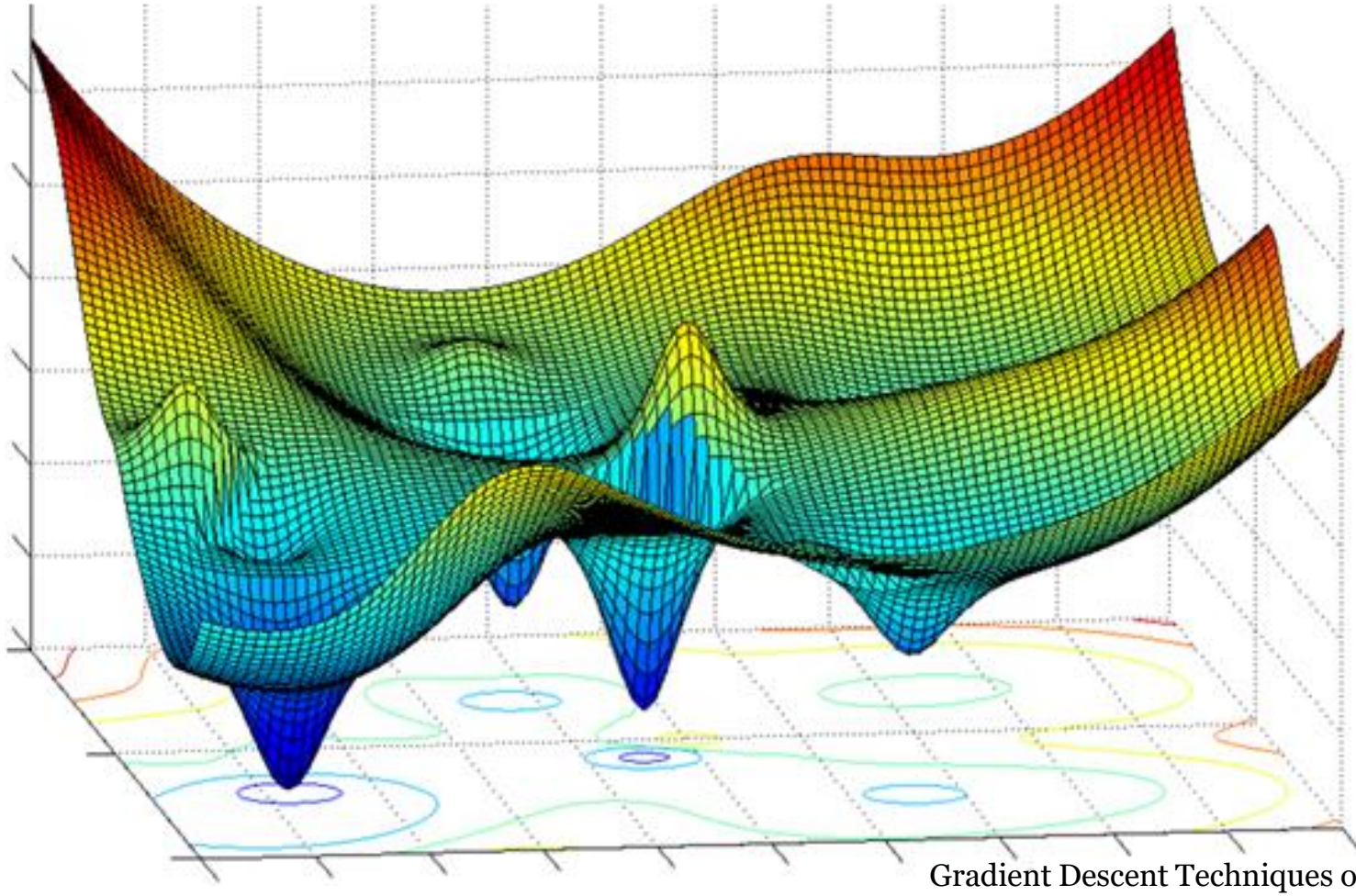
Greedy Progress: Hill Climbing

Aim: Find the global maximum



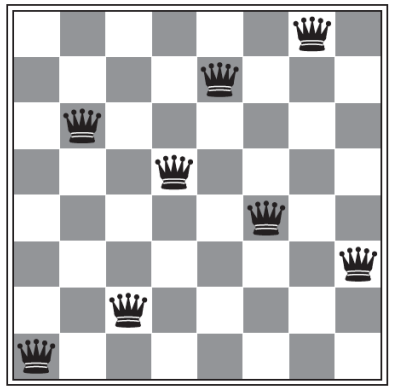
Hill Climbing: Modify the current state to try and improve it

Multi-dimensional space



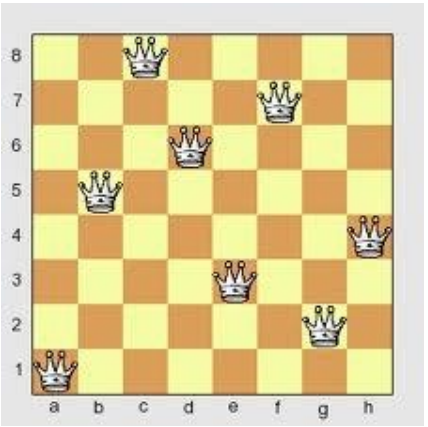
Gradient Descent Techniques often used

Hill-Climbing: 8 Queens



Problem:
Place 8 queens on a chessboard such that No queen attacks any other.

Successor Function
Return all possible states generated by moving a single queen to another square in the same column. (8*7=56)



Heuristic Cost Function
The number of pairs of queens that are attacking each other either directly or indirectly (allow intervening pieces).
Global minimum - 0

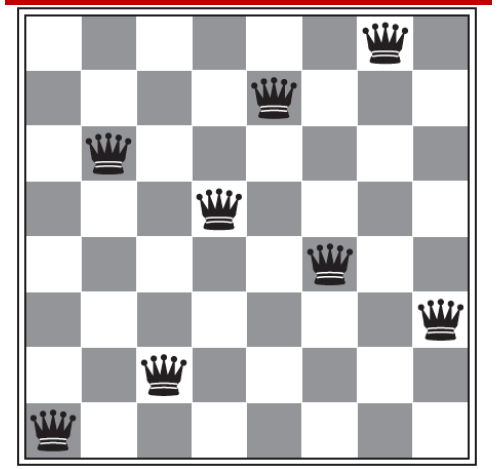
Successor state example

Current state: h=17

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

The value of h is shown for each possible successor state. The 12's are the best choices for the local move (Using steepest descent). Choose randomly on ties.

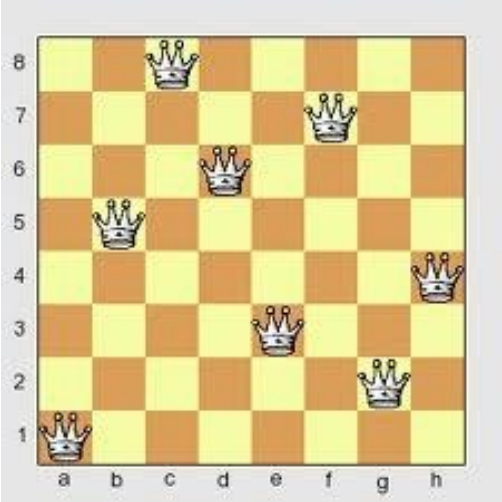
Local minimum: h=1



Global minimum h=0

Any move will increase h.

Performance



State Space: $8^8 \approx 17 * 10^6$

Branching Factor: $8 * 7 = 56$

- Starting from a random 8 queen state:
  - Steepest hill descent gets stuck *86%* of the time.
  - It is quick: average of 3 steps when it fails, 4 steps when it succeeds.
  - $8^8 \approx$17 million states!

How can we avoid local maxima, shoulders, flat maxima, etc.?

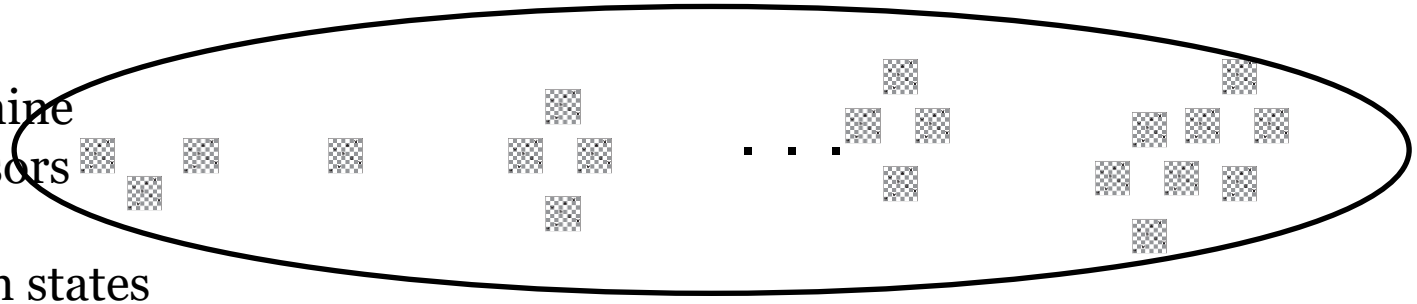
Variations on Hill Climbing

- Stochastic Hill Climbing
  - Choose among uphill moves at random, weighting choice by probability with the steepness of the move
- First Choice Hill Climbing
  - Implements stochastic hill climbing by randomly generating successors until one is generated that is better than the current state.
- Random-Restart Hill Climbing
  - Conducts a series of hill-climbing searches from randomly generated initial states until a goal is found.

Local Beam Search

Start with k random states

Determine successors of all k random states



If any successors are goal states then finished

Else select k best states from union of successors and repeat

Can suffer from lack of diversity among the k states (concentrated in small region of search space).

Stochastic variant: choose k successors at random with probability of choosing the successor being an increasing function of its value.

Simulated Annealing

Hill Climbing + Random Walk

- Escape local maxima by allowing “bad” moves (random)
  - Idea: but gradually decrease their size and frequency
  - Origin of concept: metallurgical annealing
- Bouncing ball analogy (gradient descent):
  - Shaking hard (= high temperature)
  - Shaking less (= lower the temperature)
- If Temp decreases slowly enough, best state is reached

Simulated Annealing

Gradient descent version: Minimize cost

function SIMULATED-ANNEALING($problem, schedule$) **returns** a solution state
  $current \leftarrow problem.\text{INITIAL}$
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$ / Temperature is a function of time t
    if $T = 0$ **then return** $current$
    $next \leftarrow$ a randomly selected successor of $current$
    $\Delta E \leftarrow \text{VALUE}(current) - \text{VALUE}(next)$
    if $\Delta E > 0$ **then** $current \leftarrow next$ / Descent
    else $current \leftarrow next$ only with probability $e^{-\Delta E/T}$ / Random Ascent

Boltzman Distribution

The probability decreases exponentially with the “badness” of the move - the negative amount ΔE by which the evaluation is worsened.

The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when the temperature is high, and more unlikely as T decreases.

Note: error in 4th Ed of book: $e^{-\Delta E/T}$ should be $e^{\Delta E/T}$ since ΔE is negative: Corrected in Global edition!

Some Values

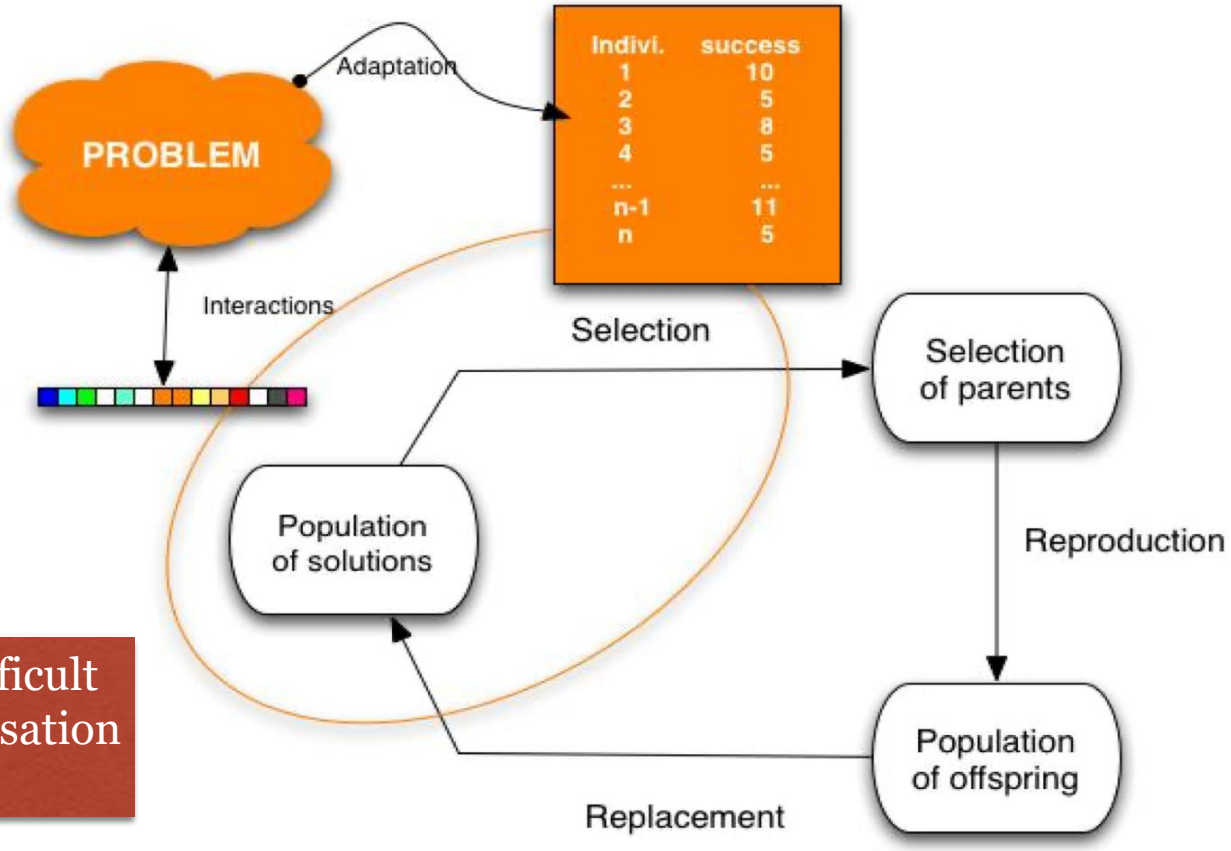
Temp:	90	80	70	60	50
ΔE	-5	-5	-5	-5	-5
$e^{\Delta E/T}$	94,59 %	93,94 %	-	-	90,48 %
ΔE	-10	-10	-10	-10	-10
$e^{\Delta E/T}$	89,48 %	88,25 %	-	-	81,87 %

Increase in badness of move ΔE

Decrease in Temperature T

Evolutionary Algorithms

Variants of Stochastic Beam Search using the metaphor of natural selection in biology

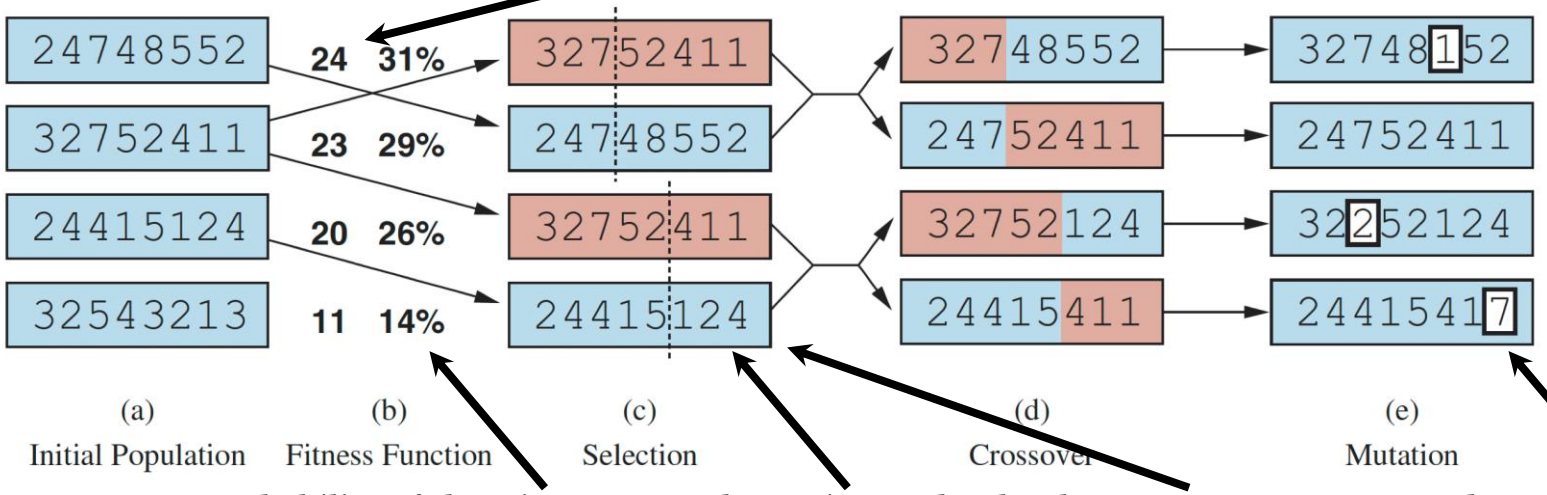


Often used for difficult non-linear optimisation problems

Genetic Algorithms

Non-attacking pairs of queens

board encodings

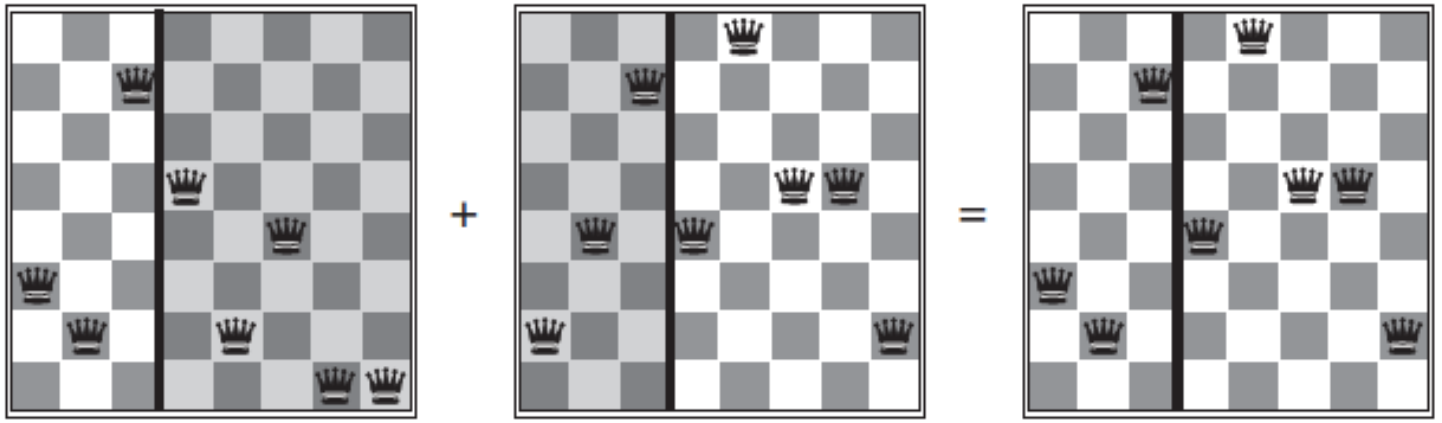


probability of choosing proportional to fitness

select pairs randomly for reproduction

choose crossover point randomly

random mutation small prob.



crossover: breed offspring in next generation

Genetic Algorithm

```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
    weights ← WEIGHTED-BY(population, fitness)
    population2 ← empty list
    for i = 1 to SIZE(population) do
      parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
      child ← REPRODUCE(parent1, parent2)
      if (small random probability) then child ← MUTATE(child)
      add child to population2
    population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Weights for individuals are computed by the fitness function
Fitness function returns # of non-attacking pairs of queens per individual

Adversarial Search

Why Study Board Games?

Board games are one of the oldest branches of AI (Shannon and Turing 1950).

- Board games present a very abstract and pure form of competition between two opponents and clearly require a form of “intelligence”.
- The states of a game are easy to represent
- The possible actions of the players are well-defined
  - Realization of the game as a search problem
  - It is nonetheless a contingency problem, because the characteristics of the opponent are not known in advance

Challenges

Board games are not only difficult because they are contingency problems, but also because the search trees can become astronomically large.

Examples:

- Chess: On average 35 possible actions from every position, 100 possible moves/ply (50 each player): $35^{100} \approx 10^{150}$ nodes in the search tree (with "only" 10^{40} distinct chess positions (states)).
- Go: On average 200 possible actions with circa 300 moves: $200^{300} \approx 10^{700}$ nodes.

Good game programs have the properties that they

- delete irrelevant branches of the game tree,
- use good evaluation functions for in-between states, and
- look ahead as many moves as possible.

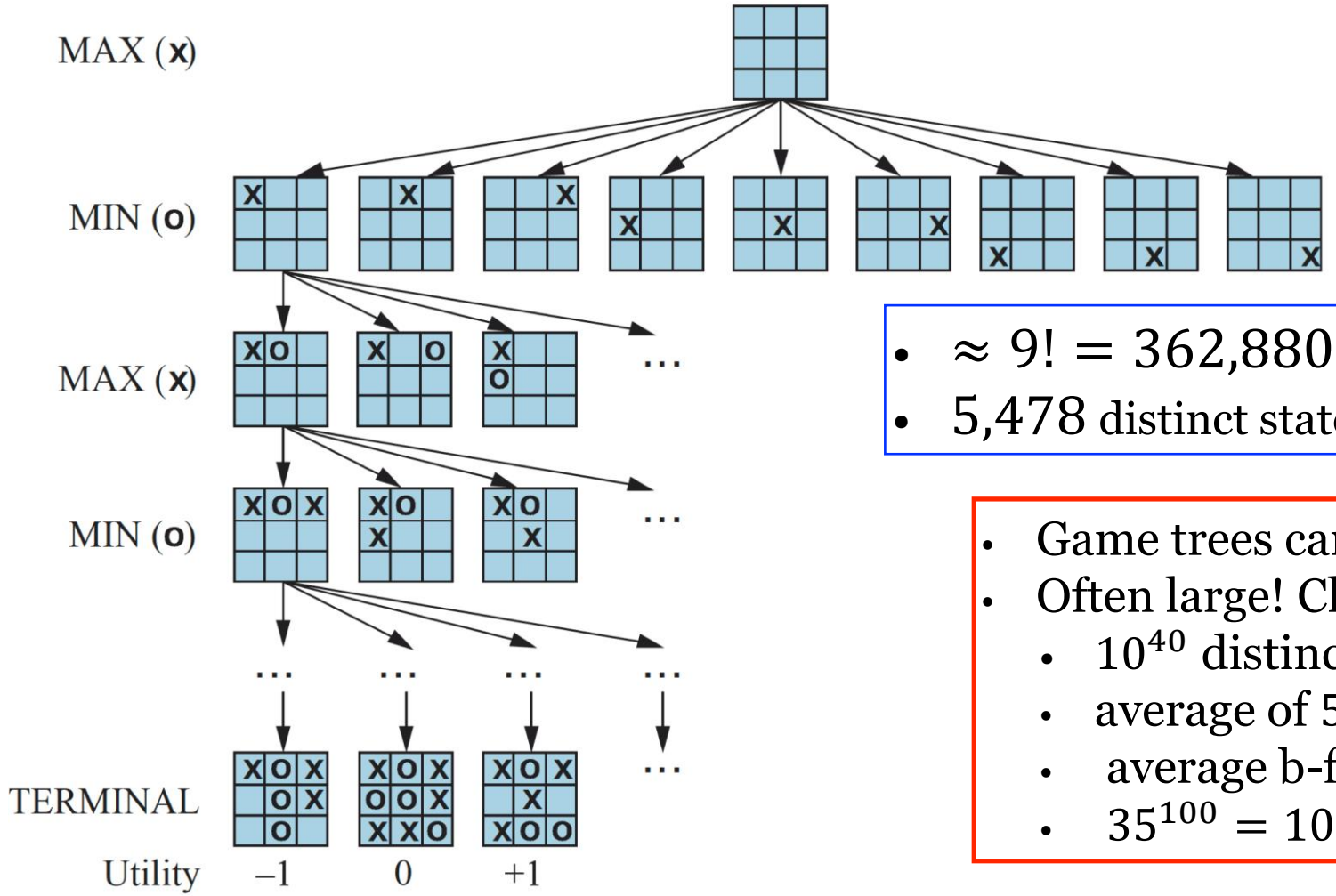
More generally: Adversarial Search

- Multi-Agent Environments
  - agents must consider the actions of other agents and how these agents affect or constrain their own actions.
  - environments can be cooperative or competitive.
  - One can view this interaction as a “game” and if the agents are competitive, their search strategies may be viewed as “adversarial”.
- Most often studied: Two-agent, zero-sum games of perfect information
  - Each player has a complete and perfect model of the environment and of its own and other agents actions and effects
  - Each player moves until one wins and the other loses, or there is a draw.
  - The utility values at the end of the game are always equal and opposite, thus the name zero-sum.
  - Chess, checkers, Go, Backgammon (uncertainty)

Games as Search

- The Game
  - Two players: One called MIN, the other MAX. MAX moves first.
  - Each player takes an alternate turn until the game is over.
  - At the end of the game points are awarded to the winner, penalties to the loser.
- Formal Problem Definition:
  - Initial State: S_0 – Initial board position
  - TO-MOVE(s) - The player whose turn it is to move in state s
  - ACTION(s) - The set of legal moves in state s
  - RESULT(s,a) - The transition model: the state resulting from taking action a in state s.
  - IS-TERMINAL(s) - A terminal test. True when game is over.
  - UTILITY(s,p) – A utility function. Gives final numeric value to player p when the game ends in terminal state s.
  - For example, in Chess: win (1), lose (-1), draw (0):

(Partial) Game Tree for Tic-Tac-Toe



- $\approx 9! = 362{,}880$ terminal nodes
- 5,478 distinct states

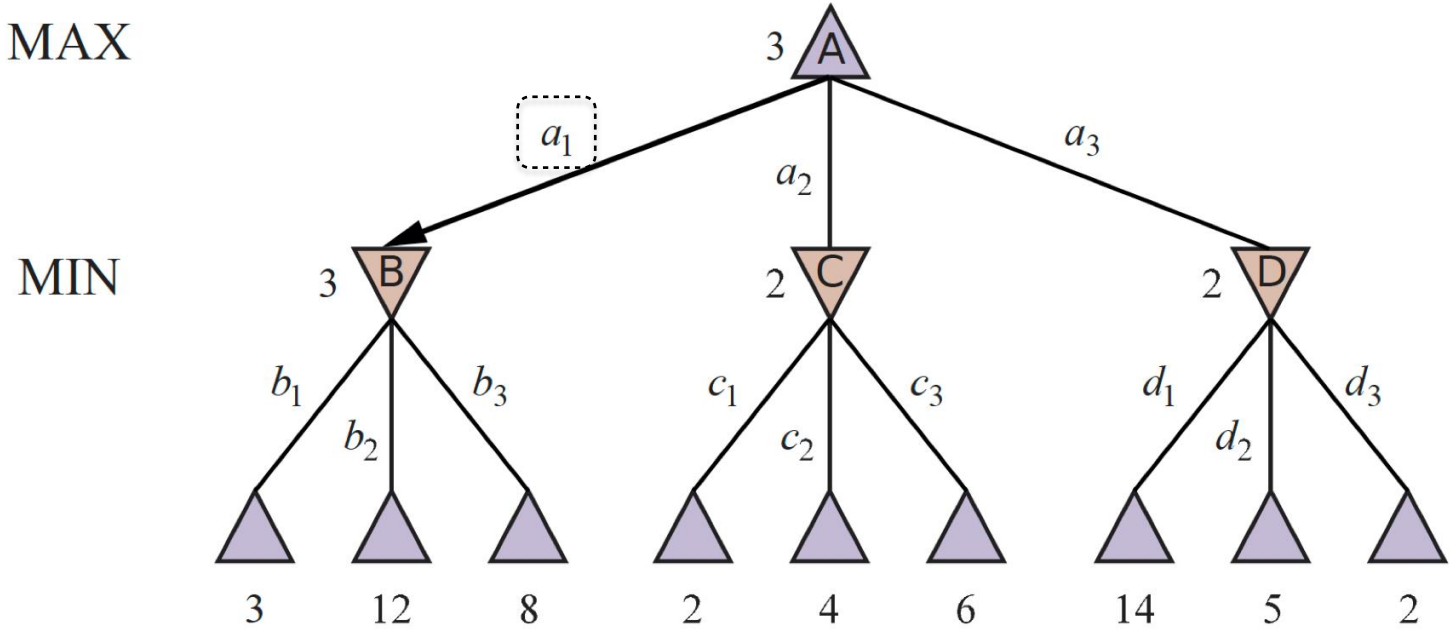
- Game trees can be infinite
- Often large! Chess has:
  - 10^{40} distinct states
  - average of 50 moves
  - average b-factor of 35
  - $35^{100} = 10^{154}$ nodes

Optimal Decisions in Games: Minimax Search

1. Generate the complete game tree using depth-first search.
2. Apply the utility function to each terminal state.
3. Beginning with the terminal states, determine the utility of the predecessor nodes (parent nodes) as follows:
   1. Node is a MIN-node
      Value is the minimum of the successor nodes
   2. Node is a MAX-node
      Value is the maximum of the successor nodes
4. From the initial state (root of the game tree), MAX chooses the move that leads to the highest value (minimax decision).

Note: Minimax assumes that MIN plays perfectly. Every weakness (i.e. every mistake MIN makes) can only improve the result for MAX.

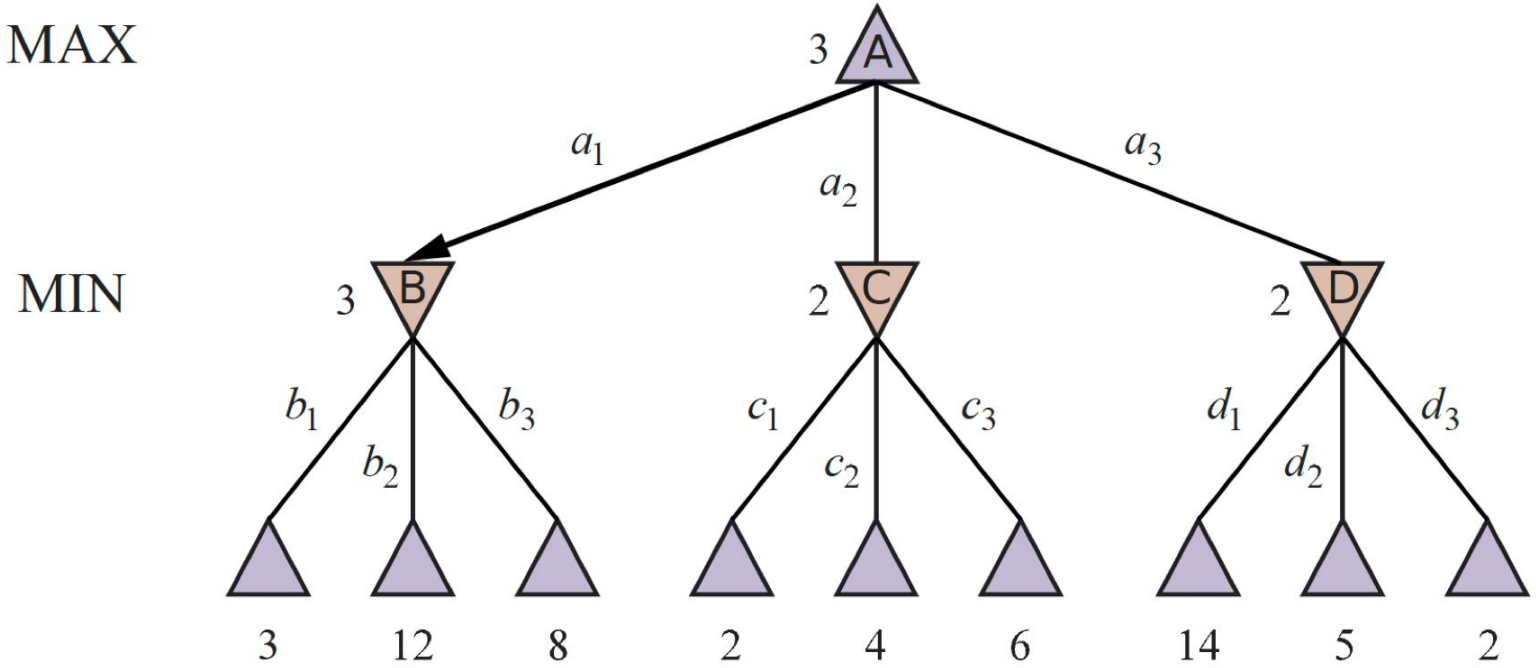
Minimax Tree



- Interpreted from MAX's perspective
- Assumption is that MIN plays optimally
- The minimax value of a node is the utility for MAX
- MAX prefers to move to a state of maximum value and MIN prefers minimum value

What move should MAX make from the Initial state?

MAX utility values



$$MINIMAX(s) =$$

$$\begin{cases} UTILITY(s, MAX) & \text{if } IS\text{-}TERMINAL(s) \\ max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

Minimax Algortihm

```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Assume max depth of the tree is m
and b legal moves at each point:

- Time complexity: $\mathbf{O}(b^m)$
- Space complexity:
  - Actions generated at same time: $\mathbf{O}(bm)$
  - Actions generated one at a time: $\mathbf{O}(m)$

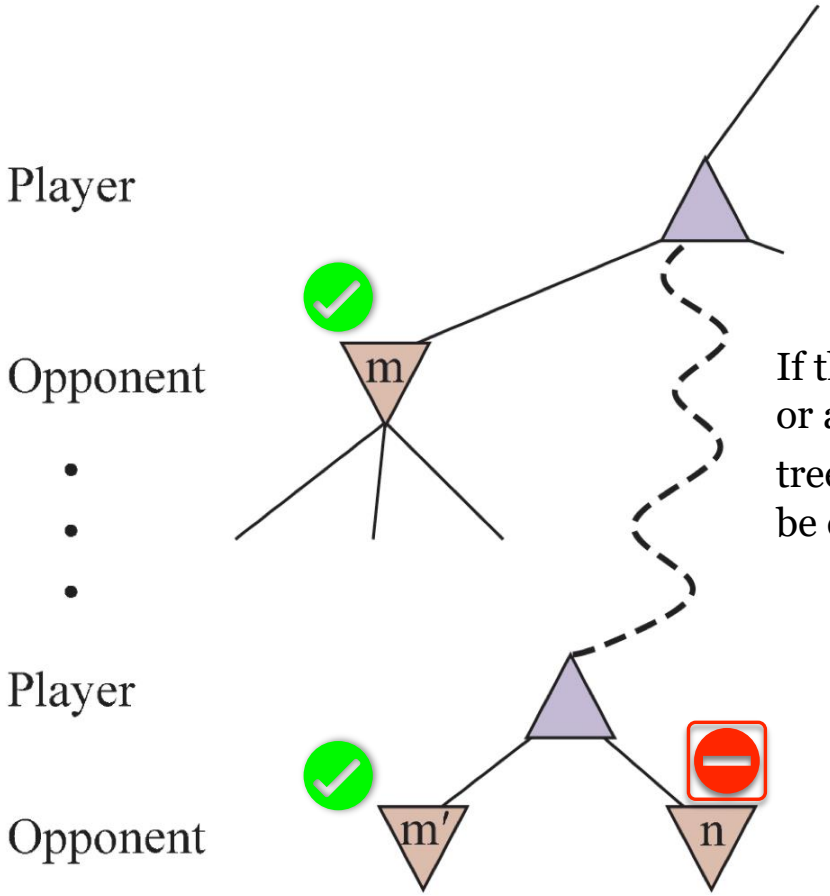
Serves as a basis for mathematical analysis of games and development of approximations to the minimax algorithm

Recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds

Alpha-Beta Pruning

- Minimax search examines a number of game states that is exponential in the number of moves (depth in the tree).
- Can be improved by using Alpha-Beta Pruning.
  - The same move is returned as minmax would
  - Can effectively cut the number of nodes visited in half (still exponential, but a great improvement).
  - Prunes branches that can not possibly influence the final decision.
  - Can be applied to infinite game trees using cutoffs.

The General idea



How do we determine when m, m' is a better choice than n?

If the player has a better choice at the same level m', or a better choice at any point higher up in the tree m, then n (and the subtree below) will never be chosen (searched)

Consider a node n somewhere in the tree
Such that the player has a choice of moving to n

Alpha-Beta Values

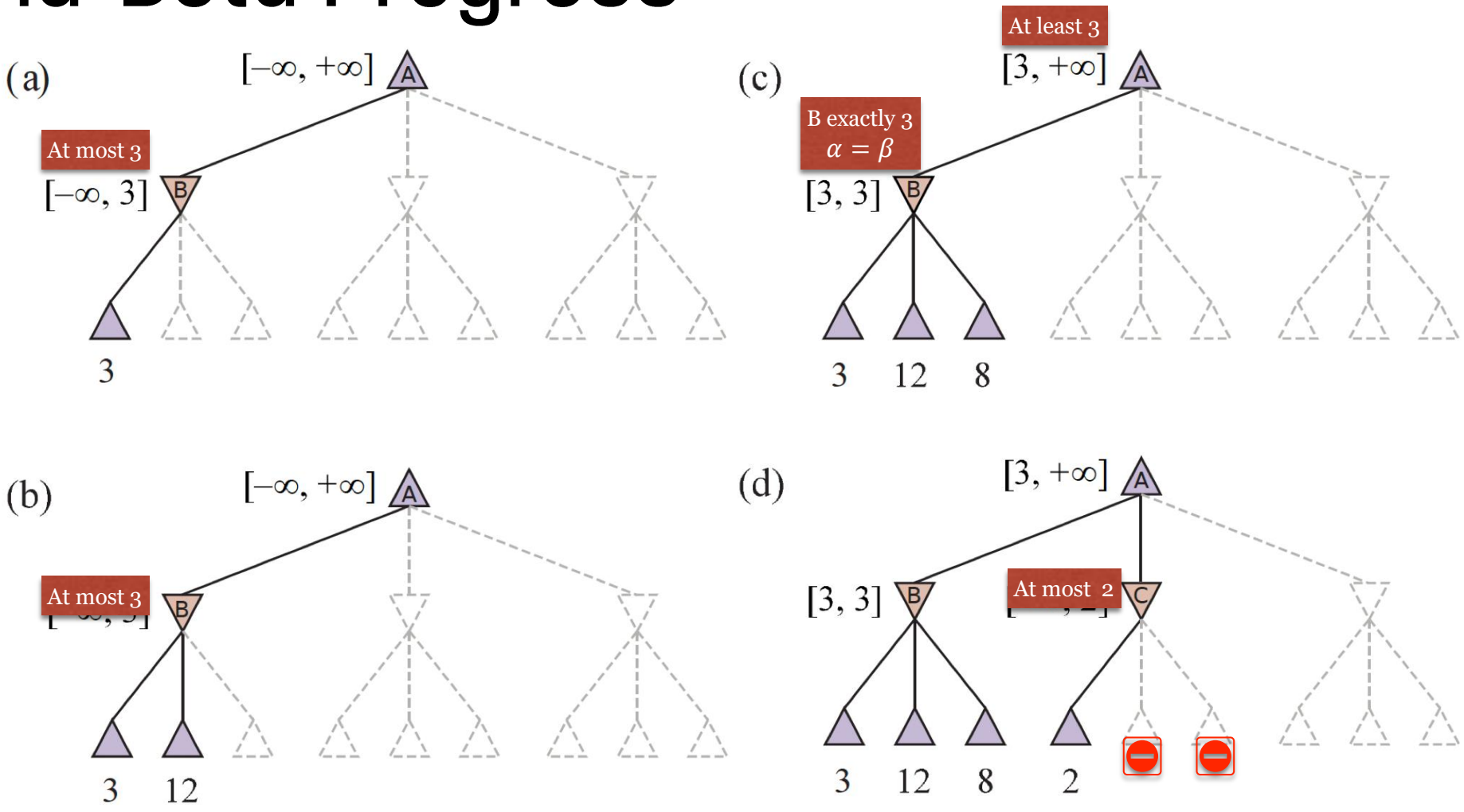
alpha – the value of the best (i.e., highest value) choice we have found so far at any choice point along the path for MAX. (actual value is at least alpha)....lower bound

beta - the value of the best (i.e., lowest value) choice we have found so far at any choice point along the path for MIN. (actual value is at most beta)...upper bound

Lower bound $[\alpha, \beta]$ Upper bound

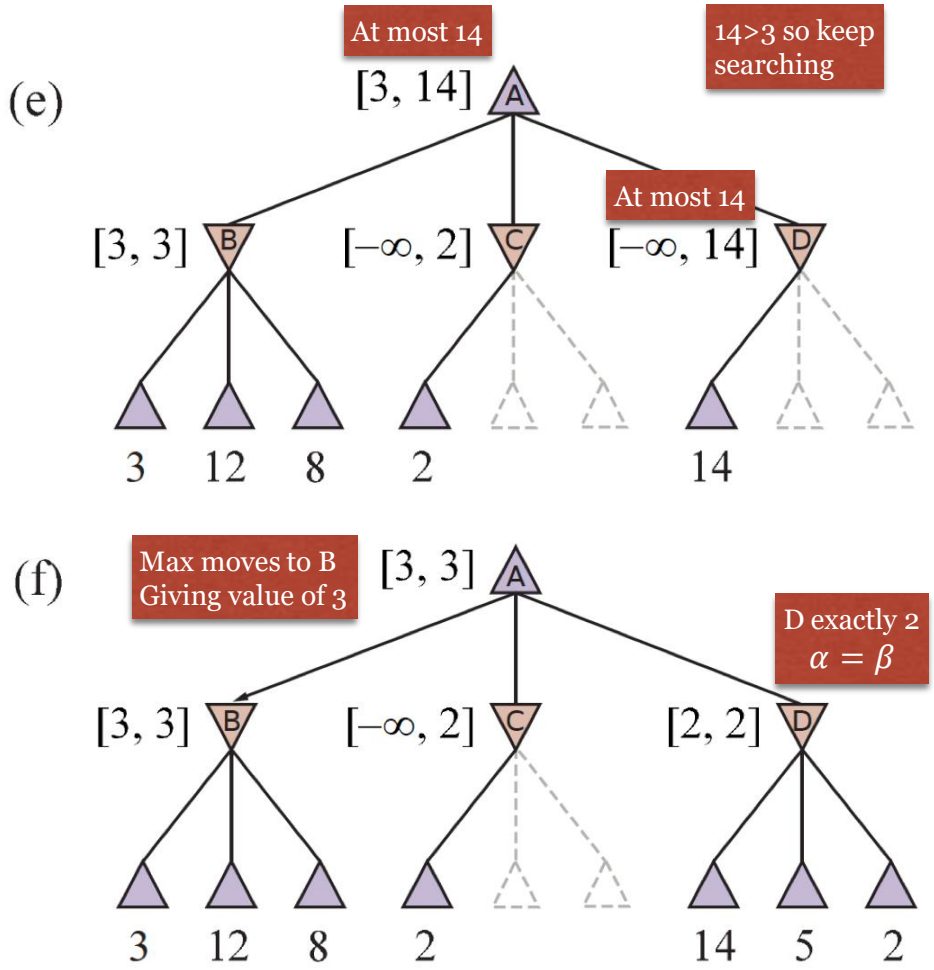
Associate lower and upper bounds
on values of nodes in the search tree

Alpha-Beta Progress



But B = 3, so MAX would never choose C
Because its value is at most 2 and could be worse
No need to search in the subtrees (terminal nodes)

Alpha-beta progress



Minimax is a depth-first search, so we only need to think of nodes/values along single paths when recursing values upwards.

2nd successor is 5
5 > 3, so keep Searching

3rd successor is 2

Alpha-Beta Search

Returns a move for MAX

Similar to Minimax search.
Functions are the same except
Bounds are maintained
on variables α and β

Effectiveness of $\alpha-\beta$ pruning is sensitive to to order in which states are examined.

With perfect move-ordering scheme, alpha-beta uses $\mathbf{O}(b^{m/2})$ nodes to pick a move rather than Minimax's $\mathbf{O}(b^m)$ nodes. But perfect move-ordering is not possible. One can get close though.

Minimax with alpha-beta pruning is still not adequate for games like chess and Go due to the huge state spaces involved. Need something better!

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Heuristic Alpha-Beta Search

Intuition:
Due to limited computation time, cutoff the search early
and apply a heuristic evaluation function to states,
Effectively treating non-terminal nodes as if they were terminal

Recall MINIMAX(s)

$$MINIMAX(s) = \begin{cases} UTILITY(s, MAX) & \text{if } IS\text{-}TERMINAL(s) \\ max_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} MINIMAX(RESULT(s,a)) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

$$H\text{-}MINIMAX(s,d) = \begin{cases} EVAL(s, MAX) & \text{if } IS\text{-}CUTOFF(s,d) \\ max_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

Heuristic Alpha-Beta Search

$$H\text{-}MINIMAX(s,d) = \begin{cases} EVAL(s,MAX) & \text{if } IS\text{-}CUTOFF(s,d) \\ max_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } TO\text{-}MOVE(s) = MAX \\ min_{a \in Actions(s)} H\text{-}MINIMAX(RESULT(s,a), d+1) & \text{if } TO\text{-}MOVE(s) = MIN \end{cases}$$

- Replace the $UTILITY(s,p)$ fn with an $EVAL(s,p)$ fn which estimates the expected utility of state s to player p.
- Replace the $IS{-}TERMINAL(s)$ test with an $IS{-}CUTOFF(s,d)$ test which must return true for terminal states, but is otherwise free to decide when to cut off the search, possibly using search depth so far or any other state properties deemed useful.

Example (Chess):

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where each f_i represents the material value of a chess piece (bishop = 3, queen=9) and the weights w_i represent how important a feature is in a state. Weights should be normalised so their sum is between range of: loss(0) to a win(+1)

Modify Alpha-Beta Search

Add bookkeeping so current depth is incremented on each recursive call

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS–CUTOFF(state, depth) then return game.EVAL(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS–CUTOFF(state, depth) then return game.EVAL(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

The Game of GO

- Two major weaknesses of Alpha-Beta Search:
  - GO has a branching factor starting at 361
    - limiting alpha-beta search to 4-5 ply (ply is a half move taken by 1 player)
  - Difficult to figure out a good evaluation function for GO
    - Material value not a strong indicator and most positions in flux until the end of the game

Modern GO programs instead use:

Monte Carlo Search (MCTS)

+ Lots of other techniques!

MCTS Strategy

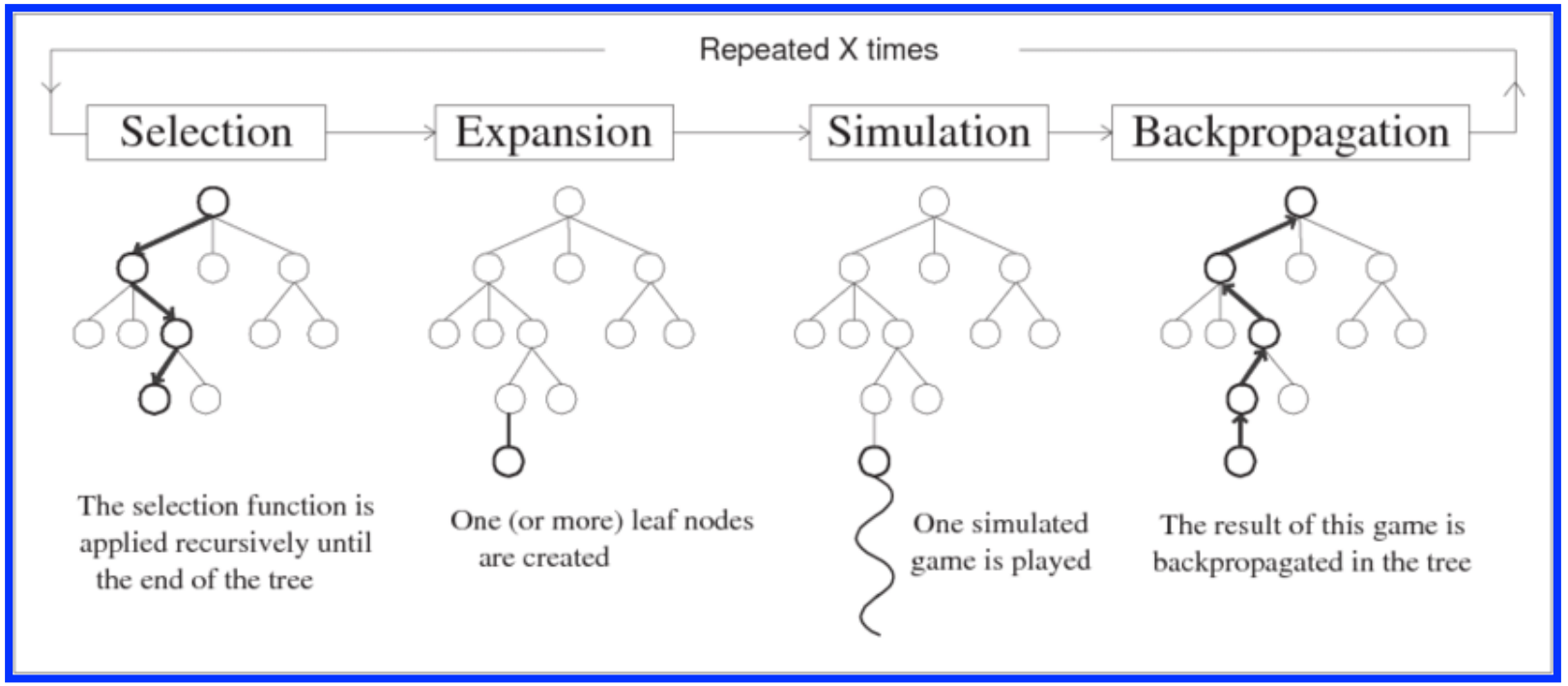
- MCTS does not use a heuristic evaluation function:
  - The value of a state is estimated as the average utility over a number of simulations of complete games starting from the state.
    - Average utility could be win percentage for example
- Simulations (also called playouts or rollouts)
  - Chooses moves first for one player and then the other until a terminal node is reached.
    - Simple policy: choose randomly
- How do we choose moves during playouts??
  - MCTS uses playout policies which are mappings between states and actions
    - Playout policies bias moves toward good ones
    - For GO and other games, playout policies can be learned from self-play using Neural Networks (Deep Learning)

MCTS Strategy

- Given a playout policy:
  - From what positions do we start the playouts?
  - How many playouts do we allocate to each position?
- Pure Monte Carlo search:
  - Do *N* simulations starting from each child in the current state of the game (determine quality of direct children (without a selection policy) and then select a move, repeat, until time runs out)
  - Focus is symmetric
  - For most games this is not adequate.
- Selection Policy selectively focuses computational resources on important parts of the game tree
  - Builds an asymmetric tree (capitalises on rich parts of search area)
  - Balances:
    - Exploitation (states that have done well in past playouts)
    - Exploration (states that have had few playouts)
  - One popular and effective selection policy is UCT (upper confidence bounds applied to trees)

4 Steps in MCTS

MCTS maintains a search tree and grows it on each iteration using the following steps:



Starting at the root of the search tree, choose a move using the selection policy, repeating the process until a leaf node is reached

Grow the search tree by generating a new child/children.

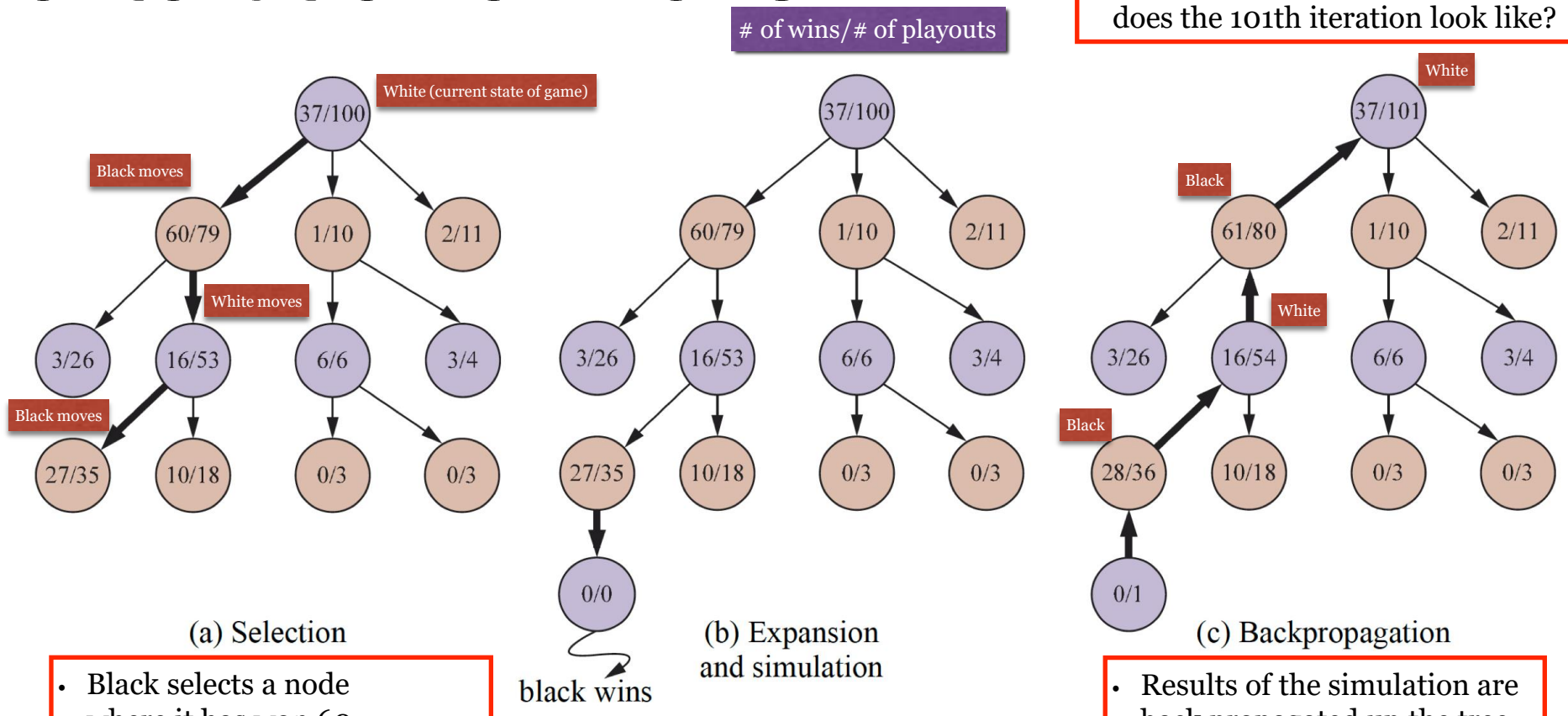
Perform a playout from a child using the playout policy. These moves are not recorded in the search tree

Use the simulation result to update the utilities of the nodes going back up to the root.

After X times: Choose the best move from start state

One Iteration of MCTS

- White has previously moved. 2024-09-12
- What should blacks move be (2nd level)?
- White has won 37 out of 100 playouts (37/100) done so far
- Suppose we will do 1000 iterations. What does the 101th iteration look like?



(a) Selection

(b) Expansion and simulation

(c) Backpropagation

- Black selects a node where it has won 60 of 79 playouts (60/79)
- Uses UCT selection metric
- Selection continues to a leaf node where black has won 27 out of 35 playouts (27/35)

- Generate a new child node labeled 0/0
- Execute a playout
- Black wins this simulation

- Results of the simulation are back propagated up the tree branch.
- Black won, so black nodes are incremented in # of wins/# of playouts
- White loses, white nodes are incremented in number of playouts only.

UCT: A Selection Policy

UCT: upper confidence bound applied to trees

Ranks each possible move based on an upper confidence bound formula called UCB1:

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

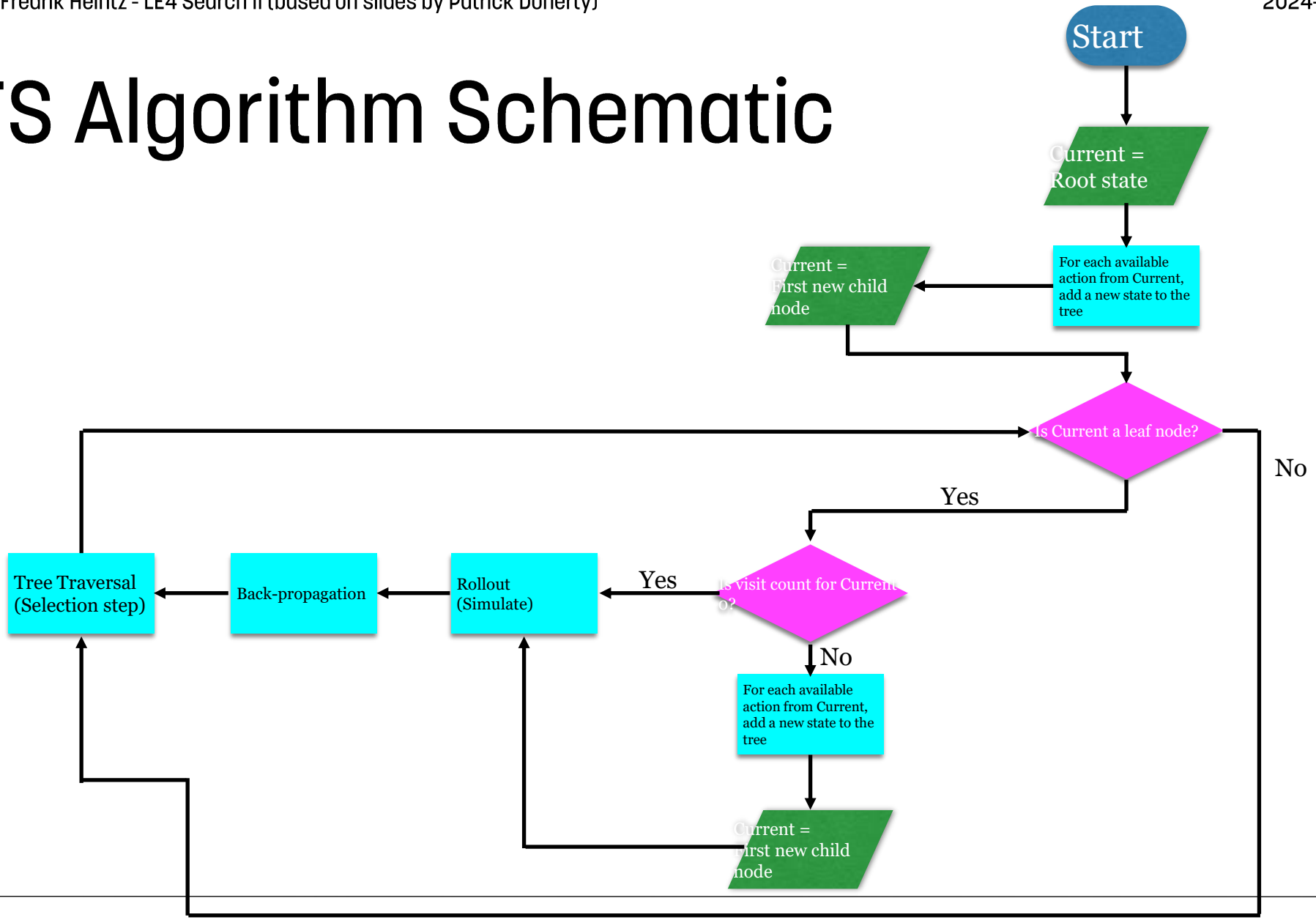
- $U(n)$: Total utility of all playouts that go through n
- $N(n)$: The number of playouts through node n
- $Parent(n)$: The parent node of node n
- $\frac{U(n)}{N(n)}$-term: is the exploitation term. The average utility of n. For example win percentage.
- $\sqrt{\ }$ - term : is the exploration term.
  - Numerator: ln of the number of times we have explored the parent
    - If n is selected some non-zero % of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with the highest average utility.
  - Denominator: count $N(n)$
    - The exploration term will be high for nodes only explored a few times
- C: Constant that balances exploitation and exploration.
  - Theoretically, $\sqrt{2}$ is best value for C, but this constant is often learned or chosen through trial and error.
  - $C = 1.4$ would choose the $60/79$ (more exploitation) node in the example during *Selection*, while $C = 1.5$ would choose the $2/11$ node (more exploration) during Selection.

MCTS Algorithm

function MONTE-CARLO-TREE-SEARCH(*state*) **returns** *an action*
  tree ← NODE(*state*)
  while IS-TIME-REMAINING() **do**
    leaf ← SELECT(*tree*)
    child ← EXPAND(*leaf*)
    result ← SIMULATE(*child*)
    BACK-PROPAGATE(*result*, *child*)
  return the move in ACTIONS(*state*) whose node has highest number of playouts

- When iterations terminate, the node with the highest number of playouts (less uncertainty) is returned rather than highest average utility.
  - The UCT/UCB1 selection strategy ensures that the node with the most playouts is almost always the node with the highest win percentage
- The time to complete a playout is linear in the depth of the game tree, so there is time for multiple playouts
  - Example: game with branching factor of 32, where average game is 100 ply:
    - Suppose we have computational power to consider a billion states before moving
      - Minimax can search 6 ply deep
      - Alpha-Beta Pruning can search 12 ply deep with perfect move ordering
      - Monte Carlo search can do 10 million playouts

MCTS Algorithm Schematic



Some Observations

ALPHAGO [2016] put four ideas together:

- Visual pattern recognition
- Reinforcement learning
- Neural networks
- Monte Carlo search

Defeated:

- Lee Sedol (by a score of 4-1 in 2015)
- Kie Jie (by a score of 3-0 in 2016)

Lee Sedol

Lee Sedol retired from Go lamenting:

"Even if I became number 1, there is an entity that can not be defeated"

Kie Jie

"After humanity spent thousands of years improvising our tactics, computers tell us that humans are completely wrong. I would go as far as to say not a single human has touched the edge of the truth of Go."

2018: ALPHAZERO surpassed ALPHAGO at Go!!

- Also defeated top programs in chess and Shogi
- Learns through self-play without human expert knowledge and without access to past games
- Uses reinforcement and deep learning

Timeline

- 1952 - Computer masters Tic-Tac-Toe
- 1994 - Computer masters Checkers
- 1997 - IBM's Deep Blue defeats Garry Kasparov in Chess
- 2011 - IBM's WATSON defeats human Jeopardy champions
- 2014 - Google's algorithms learn to play Atari Games
- 2015 - Wikipedia - " Thus it is very unlikely that it will be possible to program a reasonably fast algorithm for playing the Go endgame flawlessly, let alone the whole Go game".
- 2015 - Google's AlphaGo defeats Fan Hui (2-dan player) in Go
- 2016 - Google's AlphGo defeats Lee Sedol 4-1 (9-dan player) in Go
- 2017 - Google'sAlphaZero defeats STOCKFISH (2017 TCEC computer chess champion)
- 2018 - Google's AlphaZero surpasses AlphGo at Go (no human expertise, just self play)
- 2019 - Deep Mind's ALPHASTAR program ranks in top 0.02% of officially ranked human players for StarCraft

TDDC17 AI LE4 HT2024:
Informed Search Algorithms (Ch 3)
Search in Complex Environments (Ch 4)
Adversarial Search and Games (Ch 6)

www.ida.liu.se/~TDDC17

LINKÖPING UNIVERSITY