CPSC 481
Artificial Intelligence

Dr. Hamid Ebrahimi

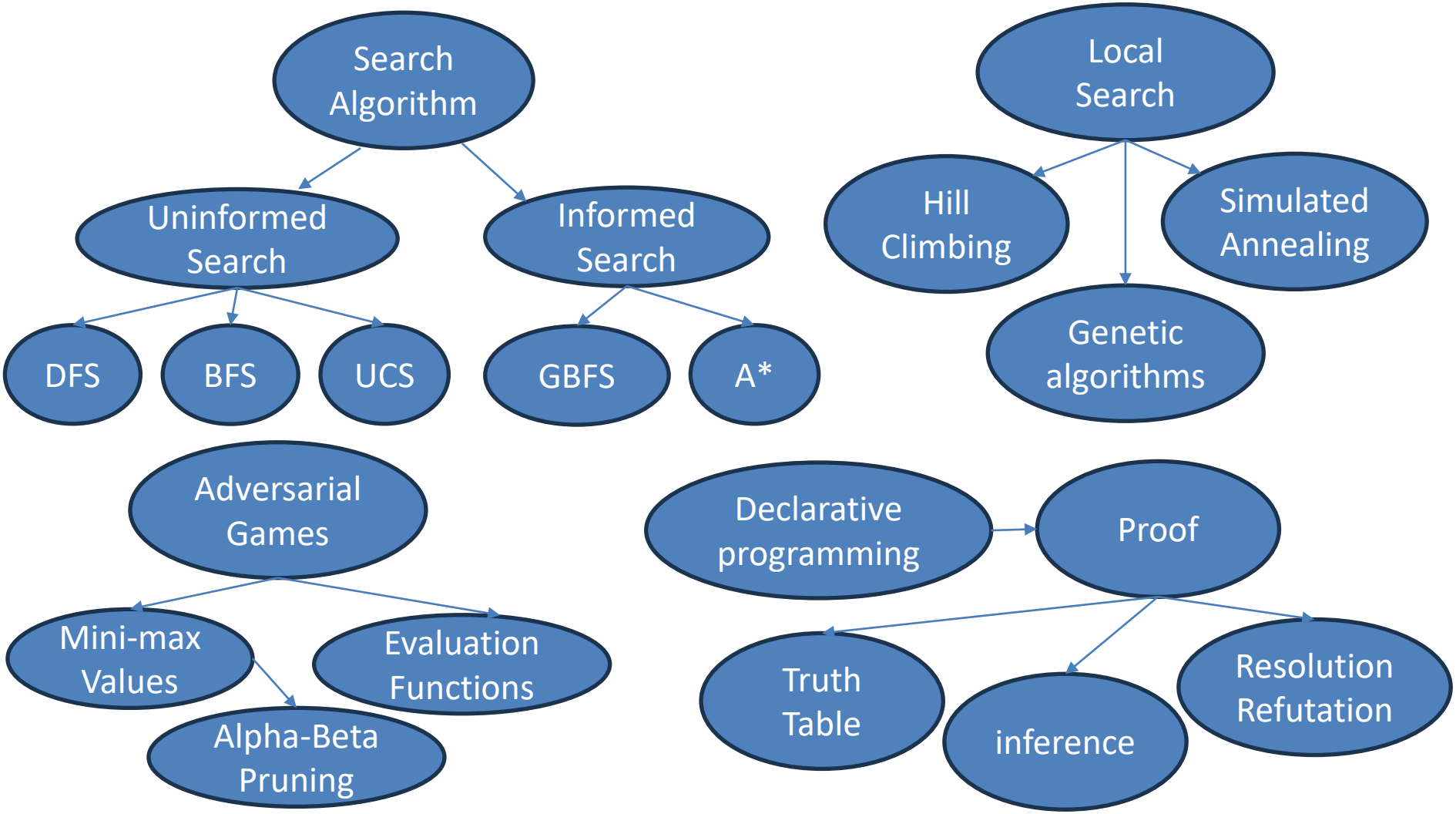
hebrahimi@fullerton.edu

What we will cover today

- Review Week 1 to Week 7

Please note: The fact that a section is not covered today does not mean it is less important or will not appear in the midterm. Due to time constraints, only selected topics are reviewed in this session.

What we covered so far!



What is AI?

- The science and engineering of making intelligent machines.
  - John McCarthy (Father of AI) – Mid 1950s coined AI
- The use of hierarchical models (like neural networks) to enable machines to learn representations of data in a way that mimics the human brain's approach to learning.
  - Geoffrey Hinton (Godfather of AI) – 2024 Nobel Prize Winner
- AI may be defined as the branch of computer science that is concerned with the automation of intelligent behavior
  - George Luger
- The study and design of intelligent agents where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success
  - Russell and Norvig

Properties of Task Environments

- **Fully observable vs. partially observable**: – Fully observable if sensors detect all aspects of environment relevant to choice of action
  - Partially observable due to noisy, inaccurate or missing sensors
  - E.g., in card games, not possible to see what cards others are holding
- **Single agent vs. multiple agents: in single agent environment, no other "thinking" entity**
  - Multiple agents: other entities whose goals depend on your own
  - Cooperative: improving other's situation, helps your own
  - Competitive: improving other's situation, worsens your own
- **Deterministic vs. stochastic**: Deterministic if the next state of the environment is completely determined by the current state and the action executed by the agent
  - Stochastic if what happens is affected by randomness
  - E.g., roll of dice in a game

Properties of Task Environments-cont.

- **Episodic vs. sequential:** episodic if a current action will not affect future actions - the agent's experience is divided into independent episodes. E.g., mail sorting system
  - Sequential if current decisions affect future decisions.
  - E.g., navigating a car is sequential, most games
- **Static vs. dynamic**: Static if the environment does not change as the agent is deciding on an action. E.g., chess
  - dynamic if the environment may change while the agent is thinking
- **Discrete vs. continuous**: in discrete environments, actions/percepts can take only specific values (e.g., chess); in continuous environments, there can be infinite such values.
  - Signals constantly coming into sensors, actions continually changing. Car driving is continuous.
- **Known vs. unknown**: In a known environment, the outcomes for all actions are given, i.e., the rules of the game are known
  - If unknown, the agent will have to learn how it works.

Properties of Task Environments-cont.

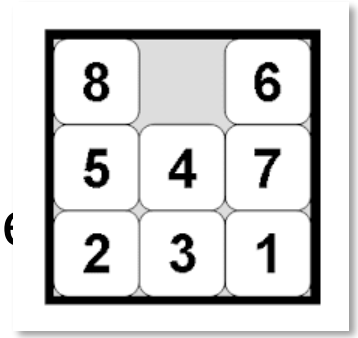
Properties	Soccer	Chutes & Ladders	CAPTCHA	CHESS	Driving a Car
Fully observable vs. partially observable?					
Deterministic vs. stochastic?					
Episodic vs. sequential?					
Static vs. dynamic?					
Discrete vs. continuous?					
Known vs. unknown?					
Single agent vs. multiple agents?					

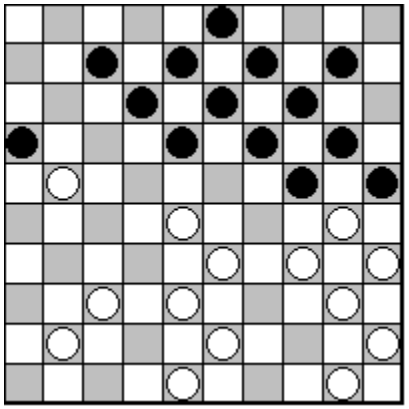
The 8-puzzle problem as state space search

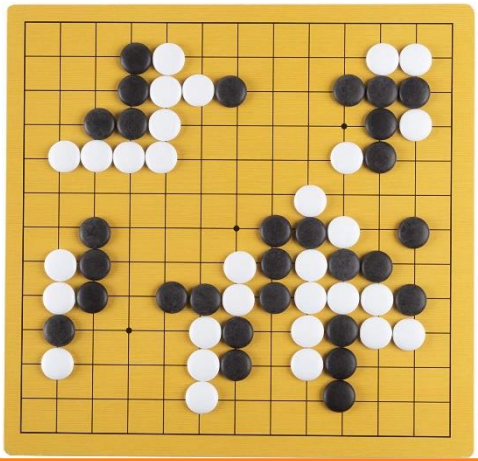
- states: possible board positions
- operators: one for sliding each square in each of four directions
  - Better, one for moving the blank square in each of four directions
- initial state: some given board position
- goal state: some given board position
- Note: the “solution” is not interesting here, we need the path.

Branching factor

- Number of next states from a given state
- Number of children of the node
- Branching factor can vary at different nodes
  - Average branching factor
- Examples:
  - 8-sliding tile puzzle?
  - 15-sliding tile puzzle?
  - Checkers?
  - Chess?
  - Go?

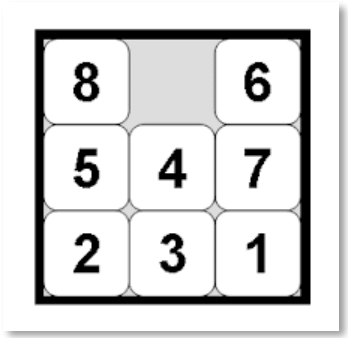


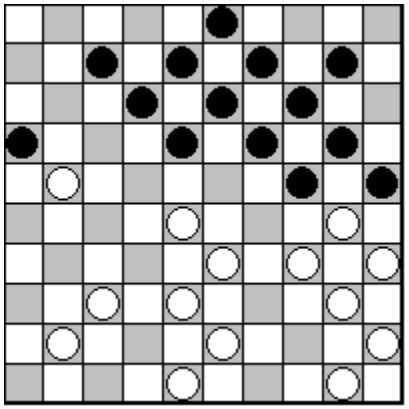


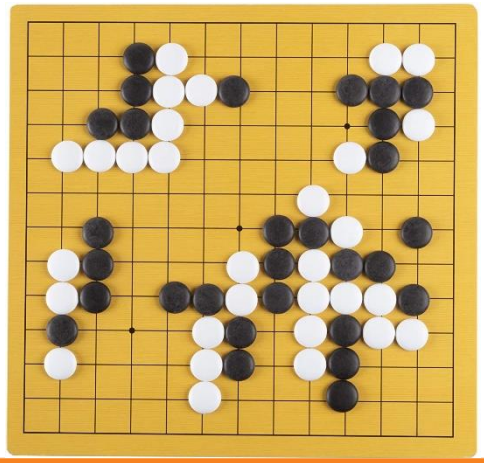


Branching factor

- Number of next states from a given state
- Number of children of the node
- Branching factor can vary at different nodes
  - Average branching factor
- Examples:
  - 8-sliding tile puzzle?
    - 2-4, average=2.13
  - 15-sliding tile puzzle?
    - 2-4
  - Checkers?
    - About 7
  - Chess?
    - About 35, average=31
  - Go?
    - 250

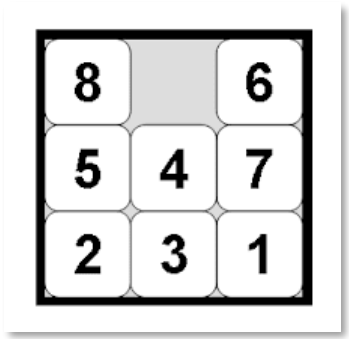


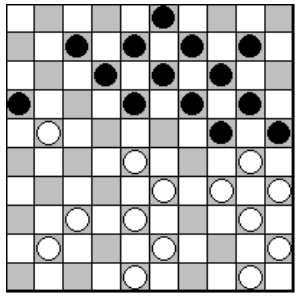


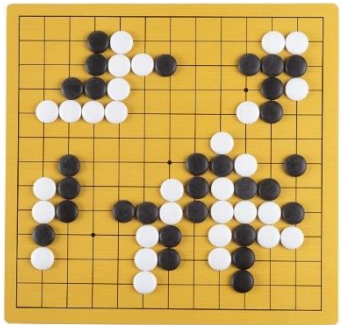


State space complexity

- State space complexity: number of states in the state space
- Examples:
  - 8-sliding tile puzzle?
  - 15-sliding tile puzzle?
  - Checkers?
  - Chess?
  - Go?
- if branching factor is 10, then
  - 10 nodes one level down from the current position
  - 10^2 (or 100) nodes two levels down
  - 10^3 (or 1,000) nodes three levels down
  - State space explosion







Searching a graph: the challenge

- Number of nodes is practically infinite
  - Cannot pre-compute all nodes and store it in a computer/data structure
  - “Search”, not a “traversal”
- Nodes can reappear in the search
  - Possible to get into loops

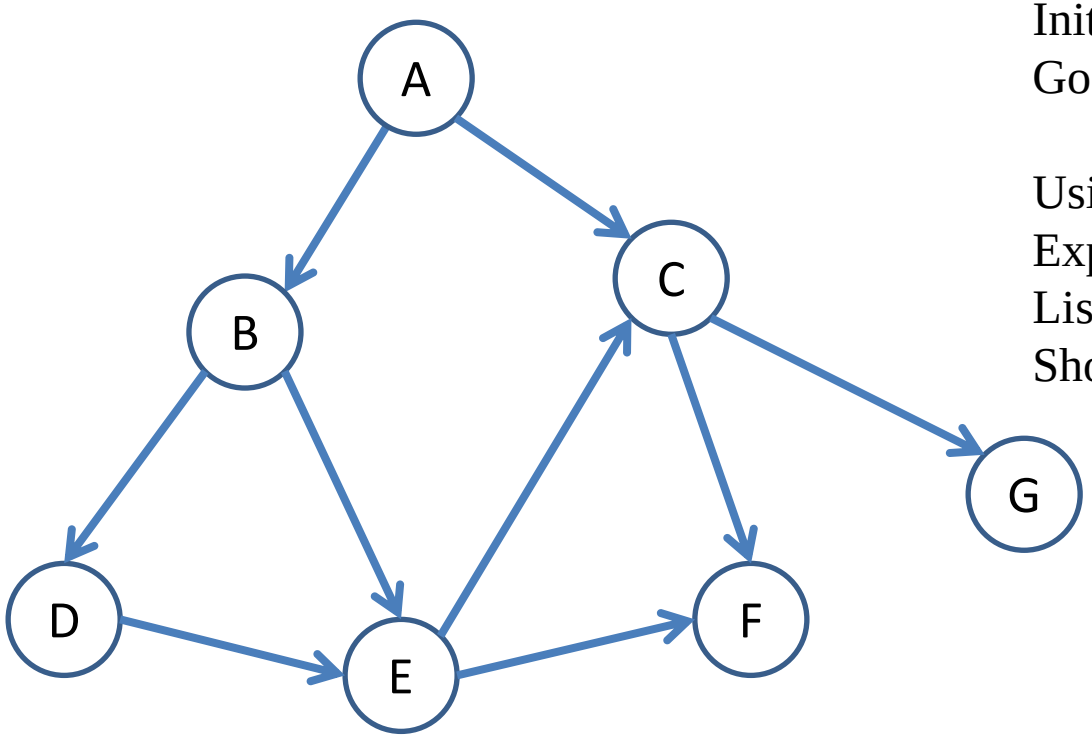
Search Algorithms

- Uninformed Search:
  - DFS
  - DFS (with loop checking)
  - BFS
  - UCS
- Informed Search
  - Greedy Best-First Search
  - A*

Depth-first TREE search

- Input: A problem
- Data structure: frontier
  - Also called “open”
  - Stack, LIFO queue

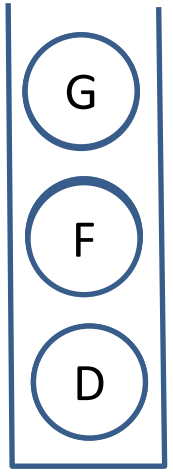
```
Initialize frontier with initial state
Loop do
    IF the frontier is empty RETURN FAILURE
    Choose top node and remove it from frontier
    IF top node is goal RETURN SUCCESS
    expand top node: pushing child nodes to the frontier
```



Initial state: A
Goal state: D

Using DFS (Tree version)
Expand child nodes in alphabetical order
List states as they are expanded
Show frontier at every step

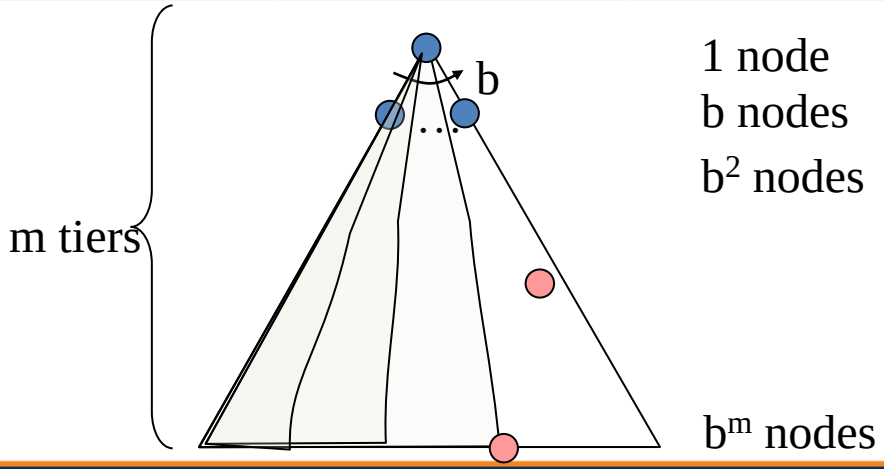
List of states:



Frontier

Searching Algorithms

Algorithm	Space Complexity	Time Complexity	Complete	Optimal
DFS	O(bm)	$O(b^m)$	No	No
DFS (with loop checking)				
BFS				
UCS				
A*				



b: branching factor,
m: number of tiers

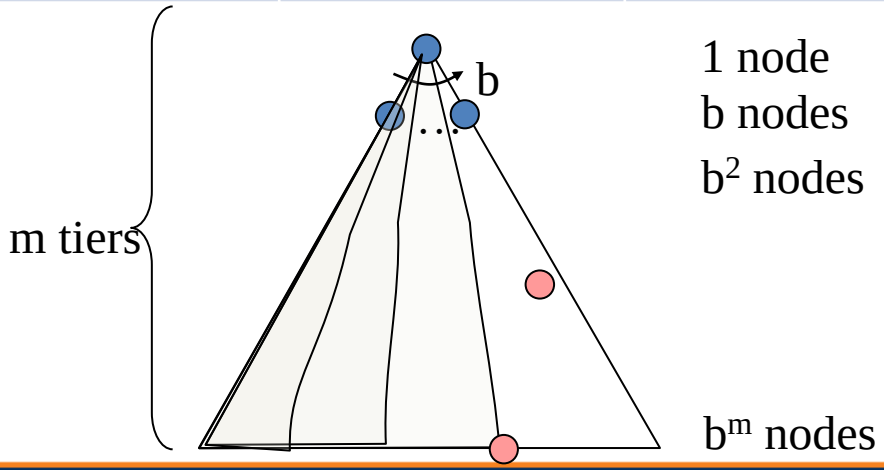
Depth-first TREE search with loop checking

- Input: A problem
- Data structures:
- frontier
  - Stack, LIFO queue
- path from root to current node
  - Parent pointers

```
Initialize frontier with initial state
Loop do
    IF the frontier is empty RETURN FAILURE
    Choose top node and remove it from frontier
    IF top node is goal RETURN SUCCESS
    expand top node
    push resulting nodes to the frontier IF they are not
    already on the path
```

Searching Algorithms

Algorithm	Space Complexity	Time Complexity	Complete	Optimal
DFS	O(bm)	$O(b^m)$	No	No
DFS (with loop checking)	O(bm)	$O(b^m)$	Yes	No
BFS				
UCS				
A*				

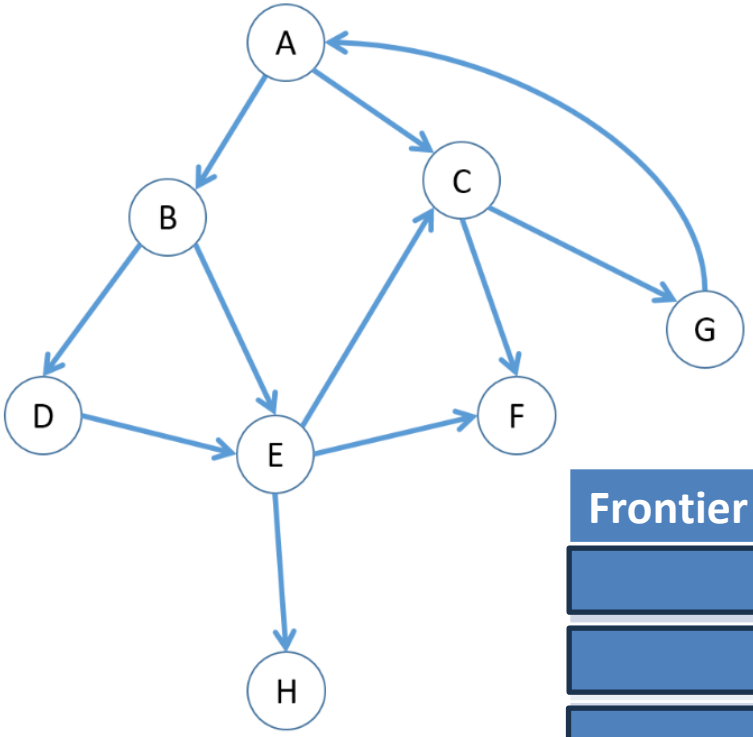


b: branching factor,
m: number of tiers

Breadth-first graph search

- Input: A problem
- Data structures:
  - Frontier (also called “open”)
    - **Queue, FIFO queue**
  - Explored (also called “closed”)
    - Set, for efficiency

```
Initialize frontier with initial state
Initialize explored to empty
Loop do
    IF the frontier is empty RETURN FAILURE
    Choose front-node from frontier and remove it
    Add front-node to explored
    FOR every child-node of front-node
      IF child-node not already on frontier or explored
        IF child-node is goal RETURN SUCCESS
        push child-node to frontier
```



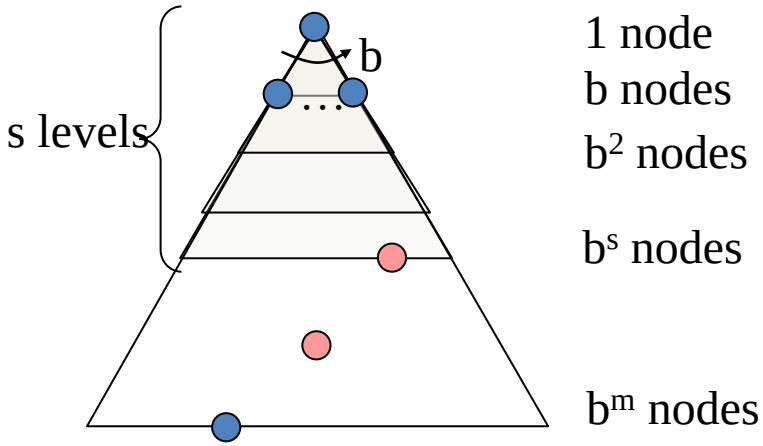
Initial state: E
Goal state: B

Frontier	Explored Set
}	

Searching Algorithms

Algorithm	Space Complexity	Time Complexity	Complete	Optimal
DFS	O(bm)	$O(b^m)$	No	No
DFS (with loop checking)	O(bm)	$O(b^m)$	Yes	No
BFS	$O(b^s)$	$O(b^s)$	Yes	Yes
UCS				
A*				

b: branching factor,
m: number of tiers
s: number of levels



Uniform cost search

- Input: A problem
- Data structures:
  - Frontier (also called “open”)
    - **Priority Queue, ordered by path cost**
  - Explored (also called “closed”)
    - Set, for efficiency

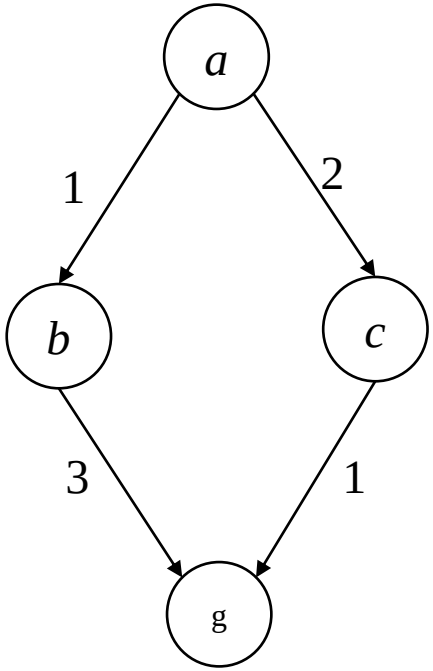
```
Initialize frontier with initial state
Initialize explored to empty
Loop do
    IF the frontier is empty RETURN FAILURE
    Choose lowest cost node from frontier and remove it
    IF lowest cost node is goal RETURN SUCCESS
    Add node to explored
    FOR every child node of node
      IF child not already on frontier or explored
        insert child node to frontier
      ELSE IF child is in frontier with higher path cost
        replace existing frontier node with child node
```

Uniform Cost Search

Strategy: expand the "cheapest" node first

Frontier is a priority queue

priority = cumulative cost



Priority Queue	Explored
A(0)	
B(1), C(2)	{A}
C(2), G(4)	{A, B}
G(3)	{A, B, C}
	Success

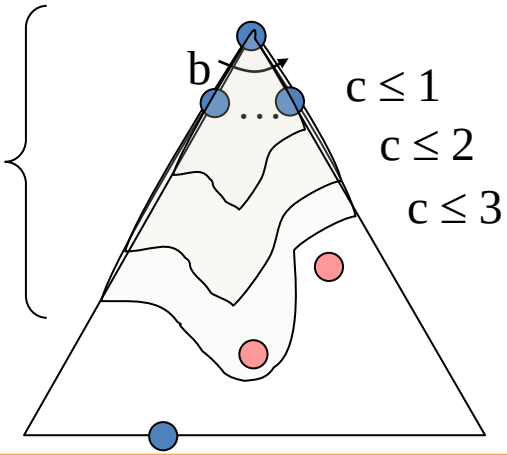
A → C → G is the least costly path with a total of 3

Searching Algorithms

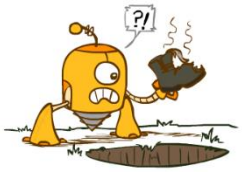
Algorithm	Space Complexity	Time Complexity	Complete	Optimal
DFS	O(bm)	$O(b^m)$	No	No
DFS (with loop checking)	O(bm)	$O(b^m)$	Yes	No
BFS	$O(b^s)$	$O(b^s)$	Yes	Yes
UCS	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$	Yes	yes
A*				

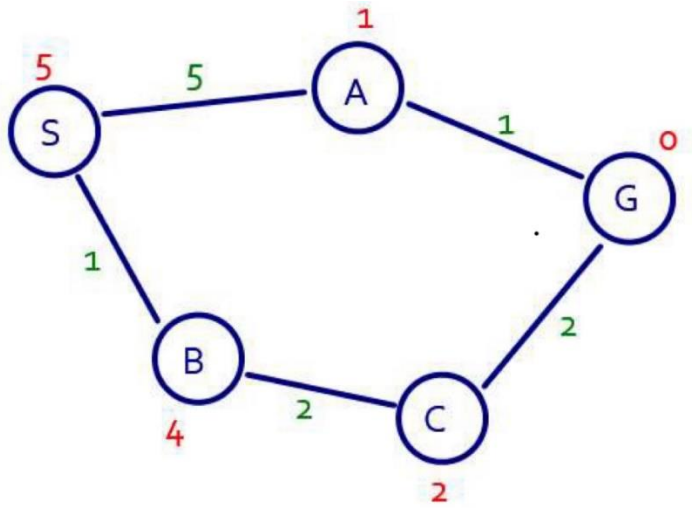
b: branching factor,
m: number of tiers
s: number of levels
C^*/ε :effective depth

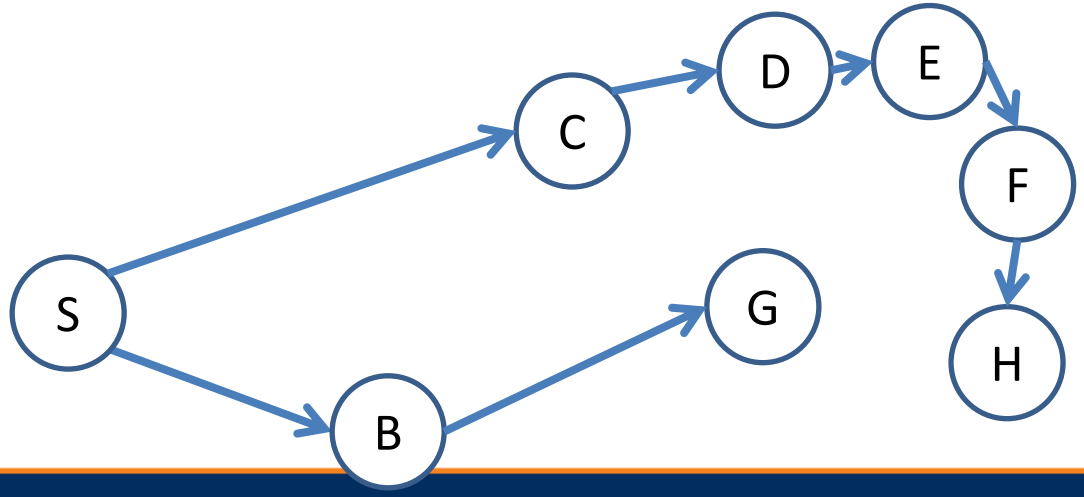
C^*/ε
"tiers"



Greedy Best-First Search

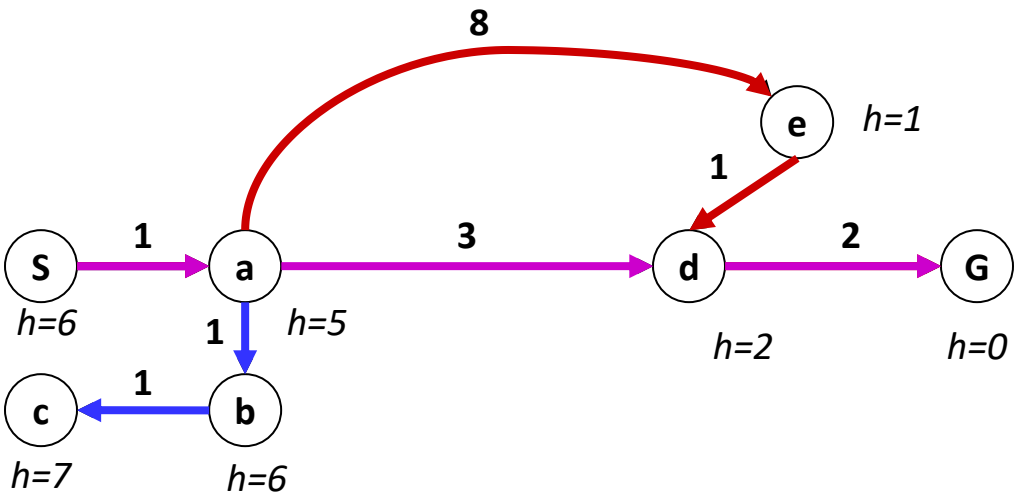


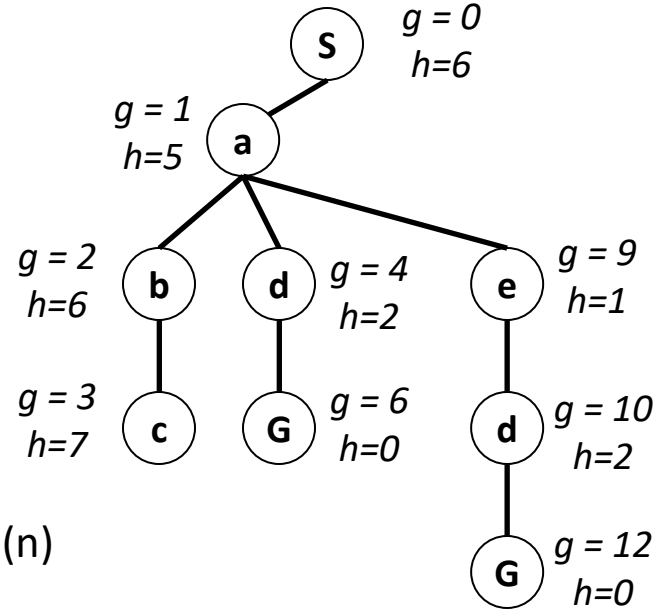




Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)





- A* Search orders by the sum: f(n) = g(n) + h(n)
- A* is pronounced "A star"

Example: Teg Grenager

A* search

- Input: A problem
- Data structures:
  - Frontier (also called “open”)
    - **Priority Queue, ordered by path cost + heuristic cost**
  - Explored (also called “closed”)
    - Set, for efficiency

```
Initialize frontier with initial state
Initialize explored to empty
Loop do
    IF the frontier is empty RETURN FAILURE
    Choose lowest cost node from frontier and remove it
    IF lowest cost node is goal RETURN SUCCESS
    Add node to explored
    FOR every child node of node
      IF child not already on frontier or explored
        insert child node to frontier
      ELSE IF child is in frontier with higher path cost
        replace existing frontier node with child node
```

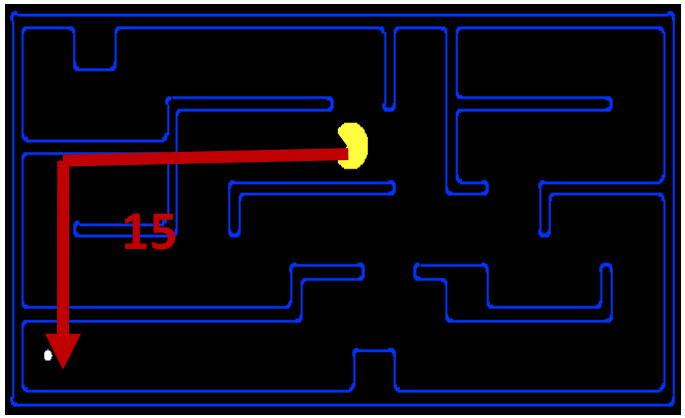
Admissible Heuristics

- A heuristic *h* is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

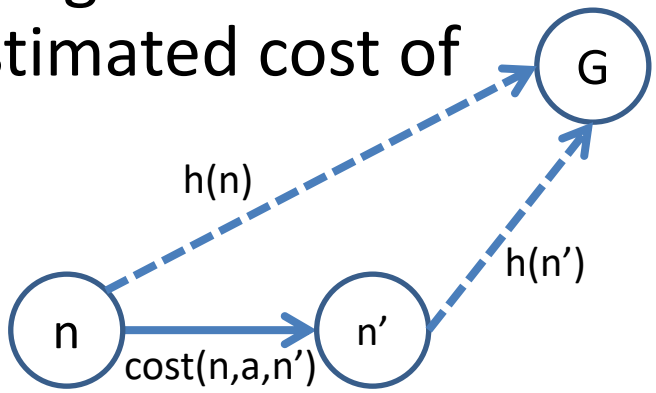
where $h^*(n)$ is the true cost to nearest goal

- Example:



Conditions for optimality of A*

2. Heuristic must be *consistent* (also called *monotonic*)
   - The estimated cost of getting to a goal is never more than cost of an action + estimated cost of next state
   - $h(n) \leq cost(n, a, n') + h(n')$
   - Triangle inequality
   - Every consistent heuristic is also admissible
   - In general, most natural admissible heuristics tend to be consistent

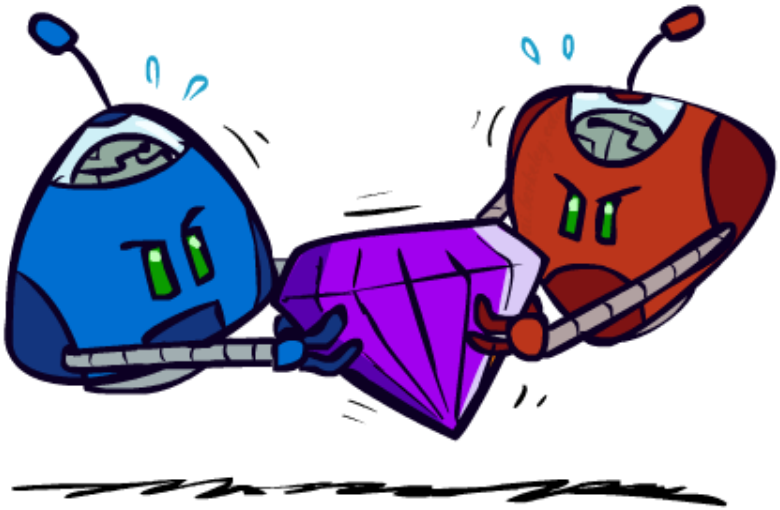


Searching Algorithms

Algorithm	Space Complexity	Time Complexity	Complete	Optimal
DFS	O(bm)	$O(b^m)$	No	No
DFS (with loop checking)	O(bm)	$O(b^m)$	Yes	No
BFS	$O(b^s)$	$O(b^s)$	Yes	Yes
UCS	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$	Yes	Yes
A*	Depends on $h^*(n) - h(n)$	Exponential	Yes	Yes

b: branching factor,
m: number of tiers
s: number of levels
C^*/ε :effective depth

Adversarial Games



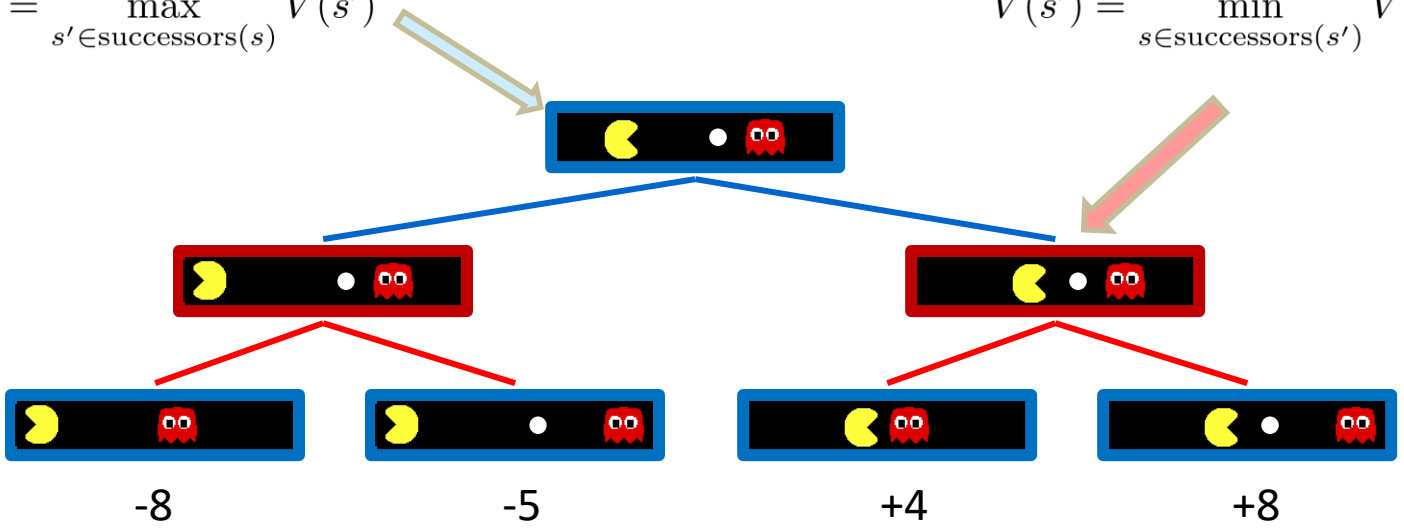
Minimax Values

States Under Agent's Control:

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

States Under Opponent's Control:

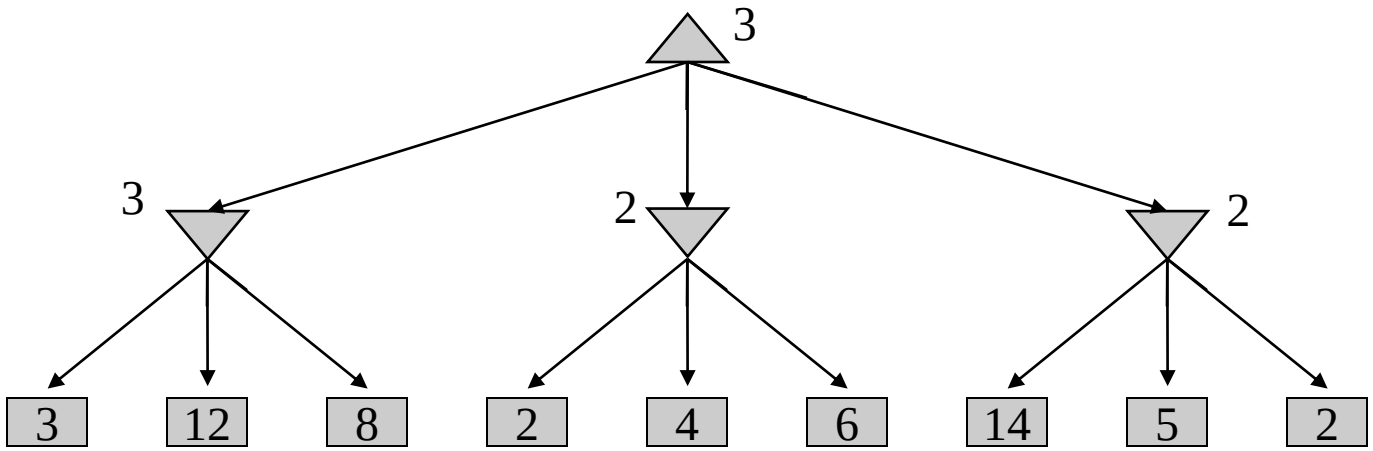
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

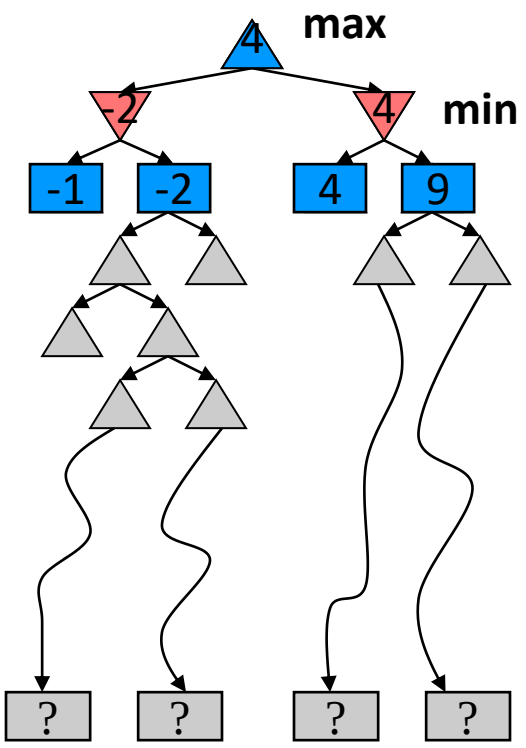
$$V(s) = \text{known}$$

Minimax Example



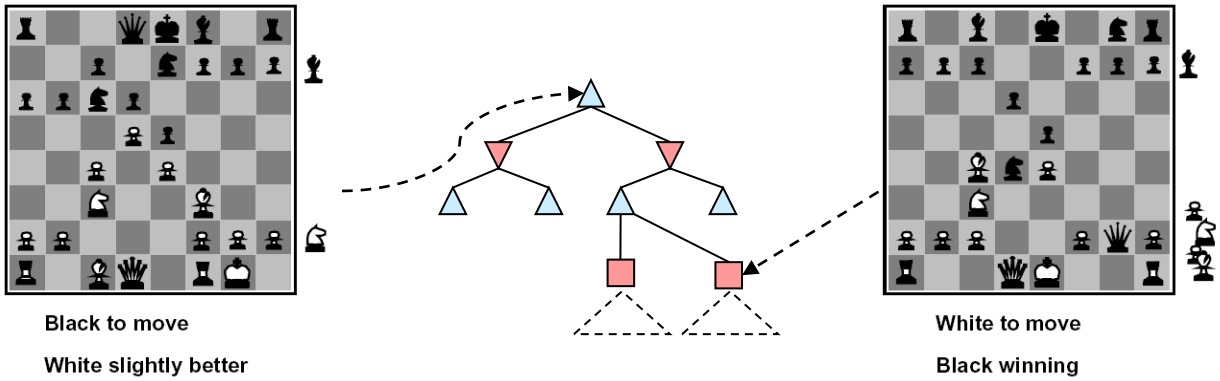
Speeding up minimax

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an **evaluation function** for non-terminal positions
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



Evaluation Functions

- Definition: An evaluation function is a (possibly very weak) estimate of the minimax value of an internal node
- Evaluation functions score non-terminals in depth-limited search

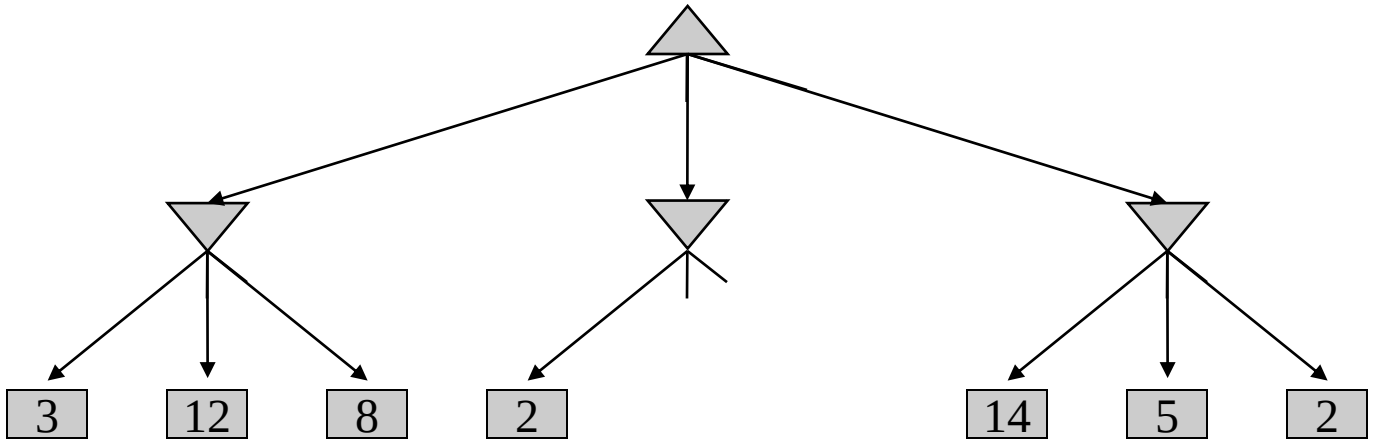


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

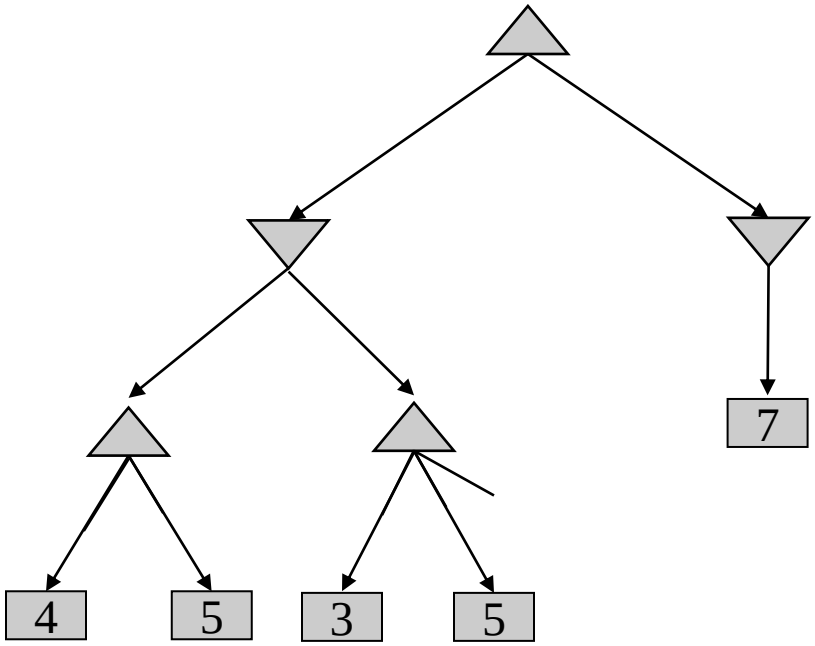
- e.g. $f_1(s)$ = 9*(QW – QB) + 5*(RW – RB) + 3*(KW – KB) + 3*(BW – BB) + 1*(PW – PB)

Minimax Pruning (α)



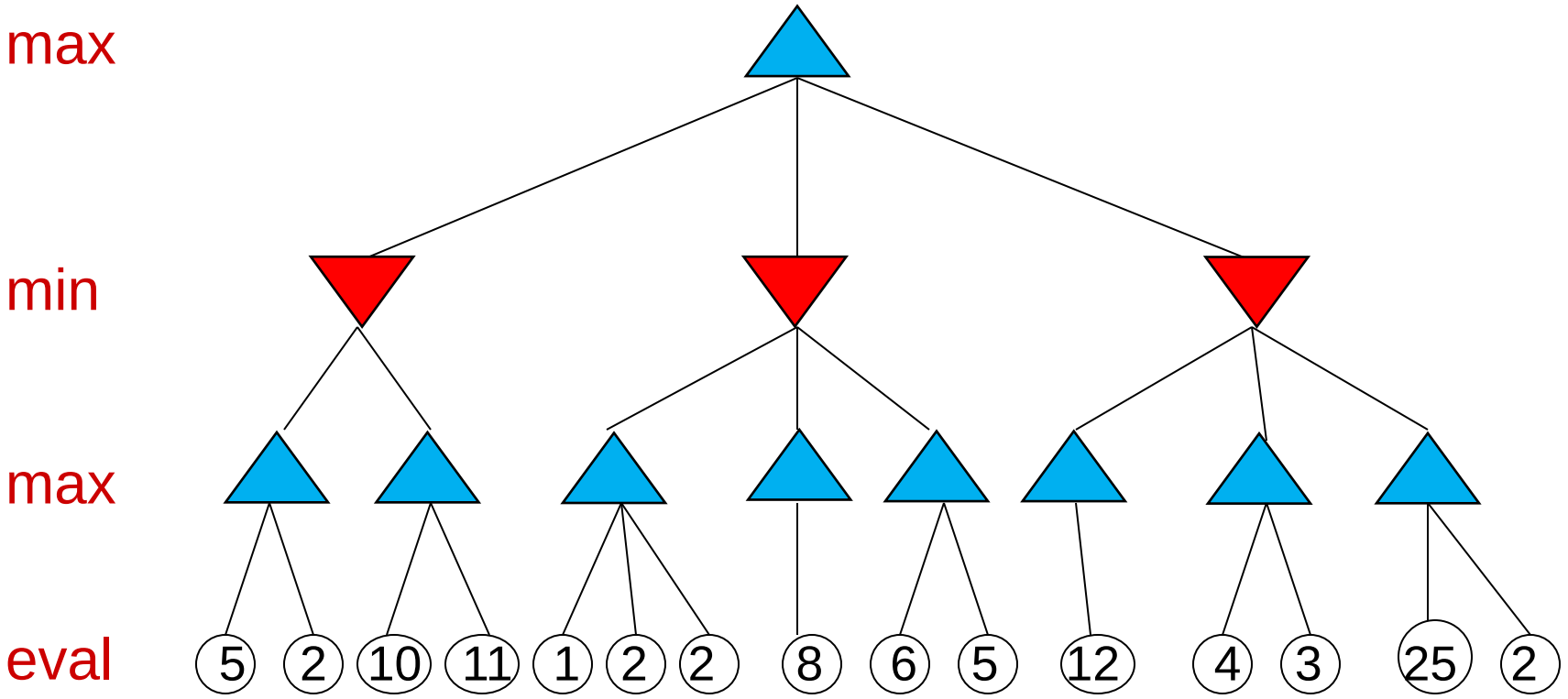
α: MAX's best option on path to root

Minimax Pruning (β)



β: MIN's best option on path to root

Alpha-Beta Pruning Practice



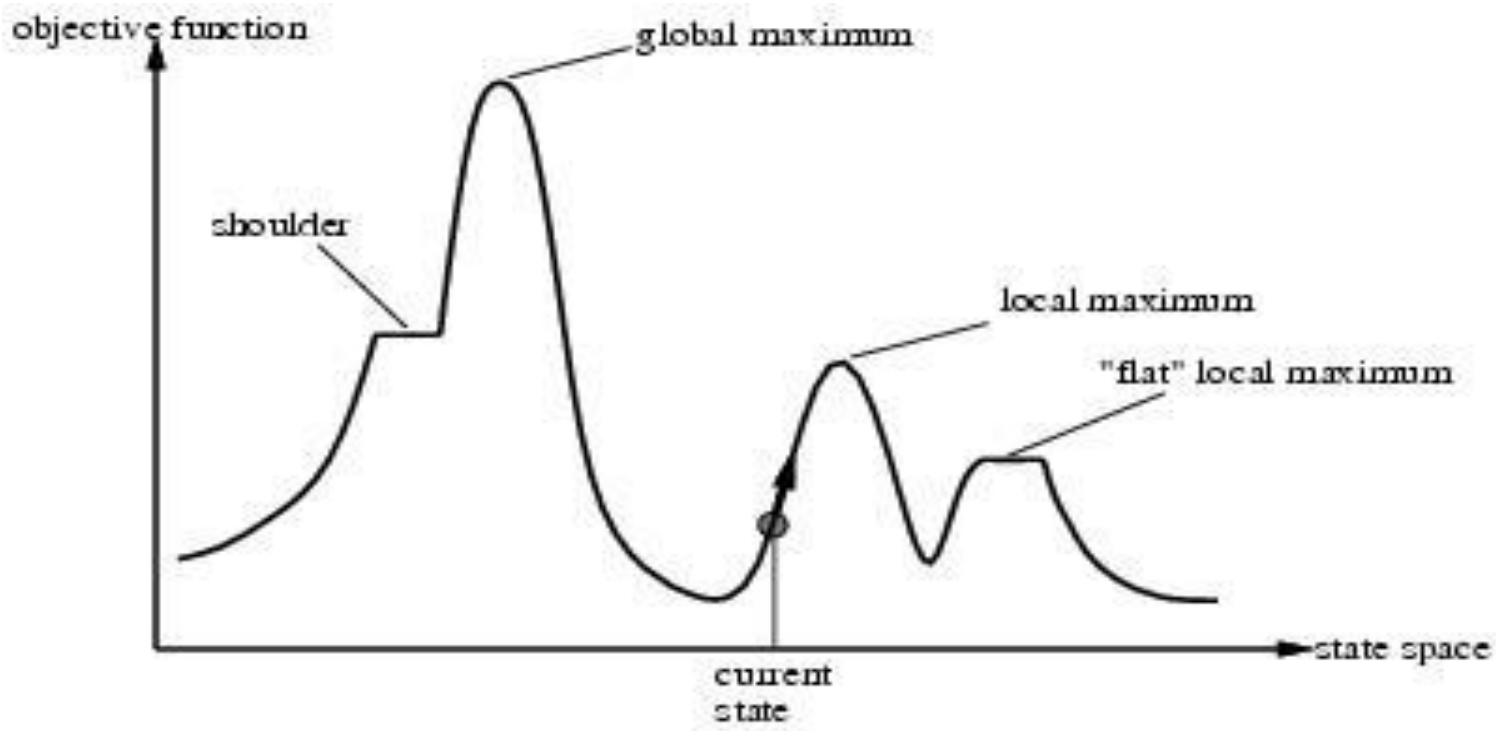
Local Search (Path vs. Goal)

- Path: BFS/DFS/UCS/A*
- Goal:
  - Optimization problems and Local search algorithms
    - Hill climbing search
    - Simulated annealing
    - Genetic algorithms

Local search

- Challenge:
  - Very large number of states
- Key idea:
  - Keep track of a "current state"
    - And its objective function value
  - Consider only successors of current state
    - Successor of a state: a state that is only "slightly different" from that state (also called neighbor)
    - If a successor is better than current state (higher objective function value), replace current state with this successor
  - Ignore paths
  - Repeat

Hill-climbing



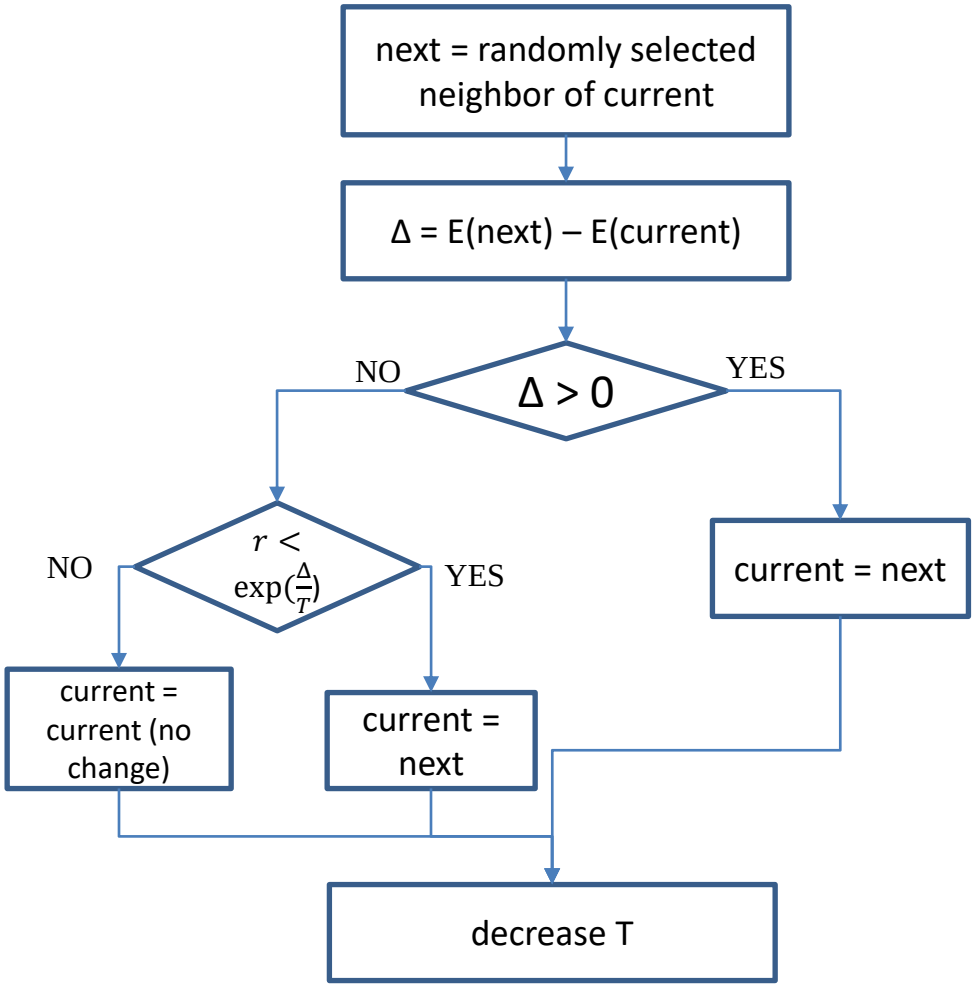
Local maxima
gets stuck when it reaches a state that has no neighboring states that are better.
Shoulder
all direct successors are not better; but a close neighbor is
Plateau
All neighboring states are not better; how far away should I keep looking for a better state?

Simulated Annealing

- Simulated Annealing = hill-climbing with **stochastic** moves

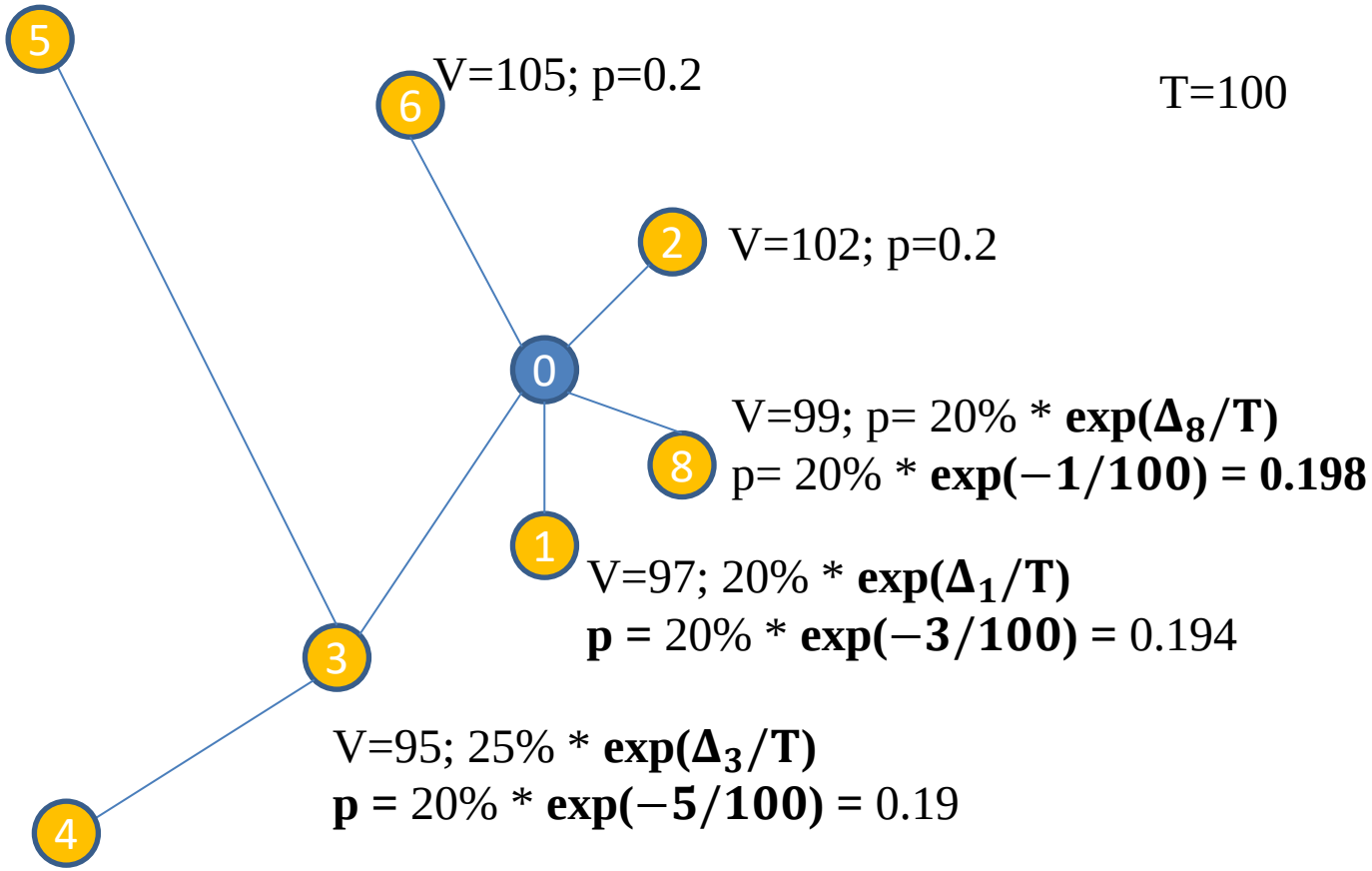
Basic ideas:

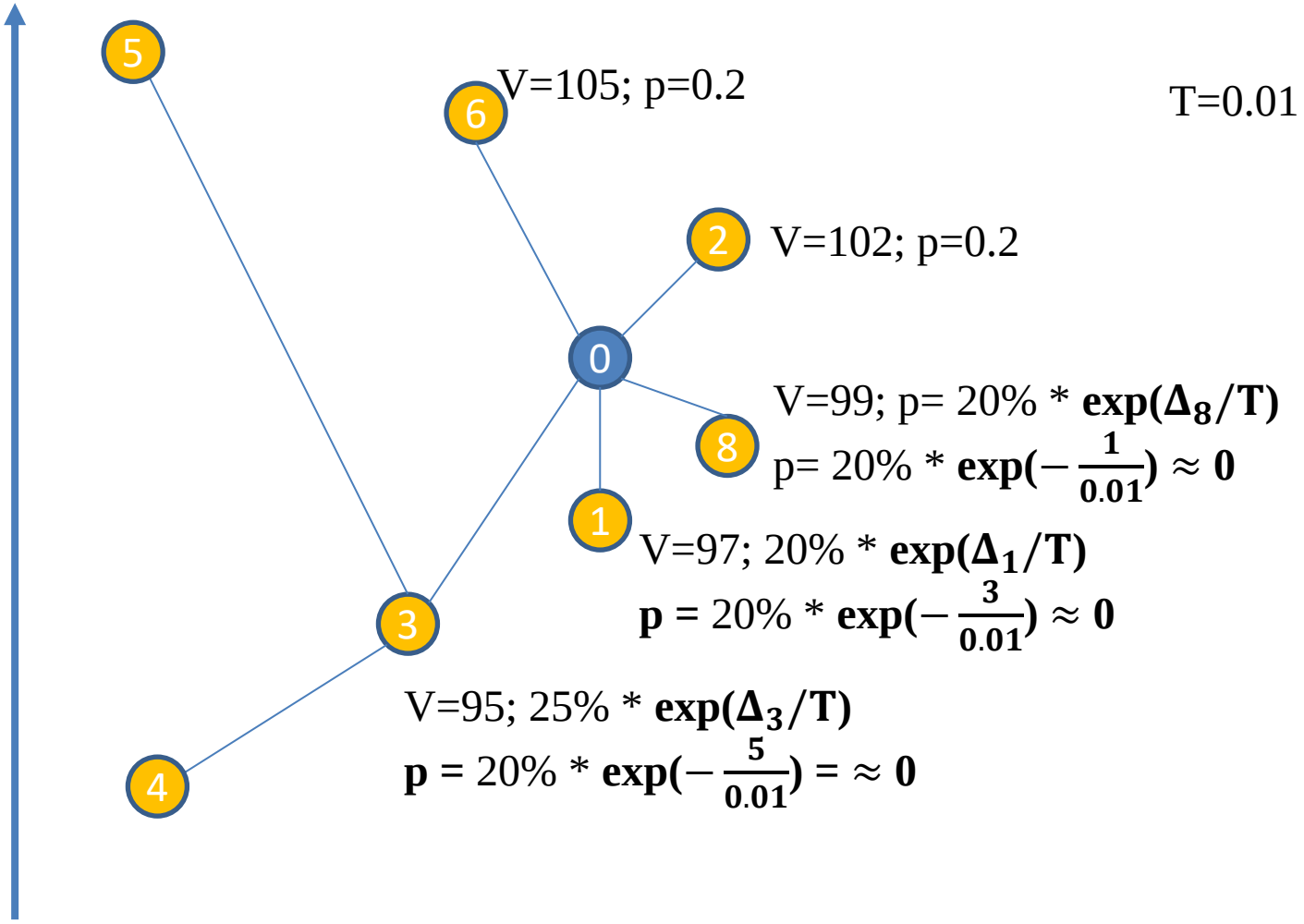
- like hill-climbing but instead of picking the best move, pick one **randomly**
- Δ = change in objective function value
- if Δ is positive, then move to that state
- Otherwise:
  - move to this state with **probability proportional to Δ**
  - thus: worse moves (large negative Δ) are executed less often
- There is always **a chance of escaping** from local maxima over time



Class work

- Let current state value be 100
- Has 4 neighbors with values: 102, 105, 95, 97, 99
- Calculate the probability of
  - Moving to 102
  - Moving to 105
  - Moving to 95
  - Moving to 97
  - Moving to 99
  - Staying at 100
- Repeat for T=0.01 and T=100

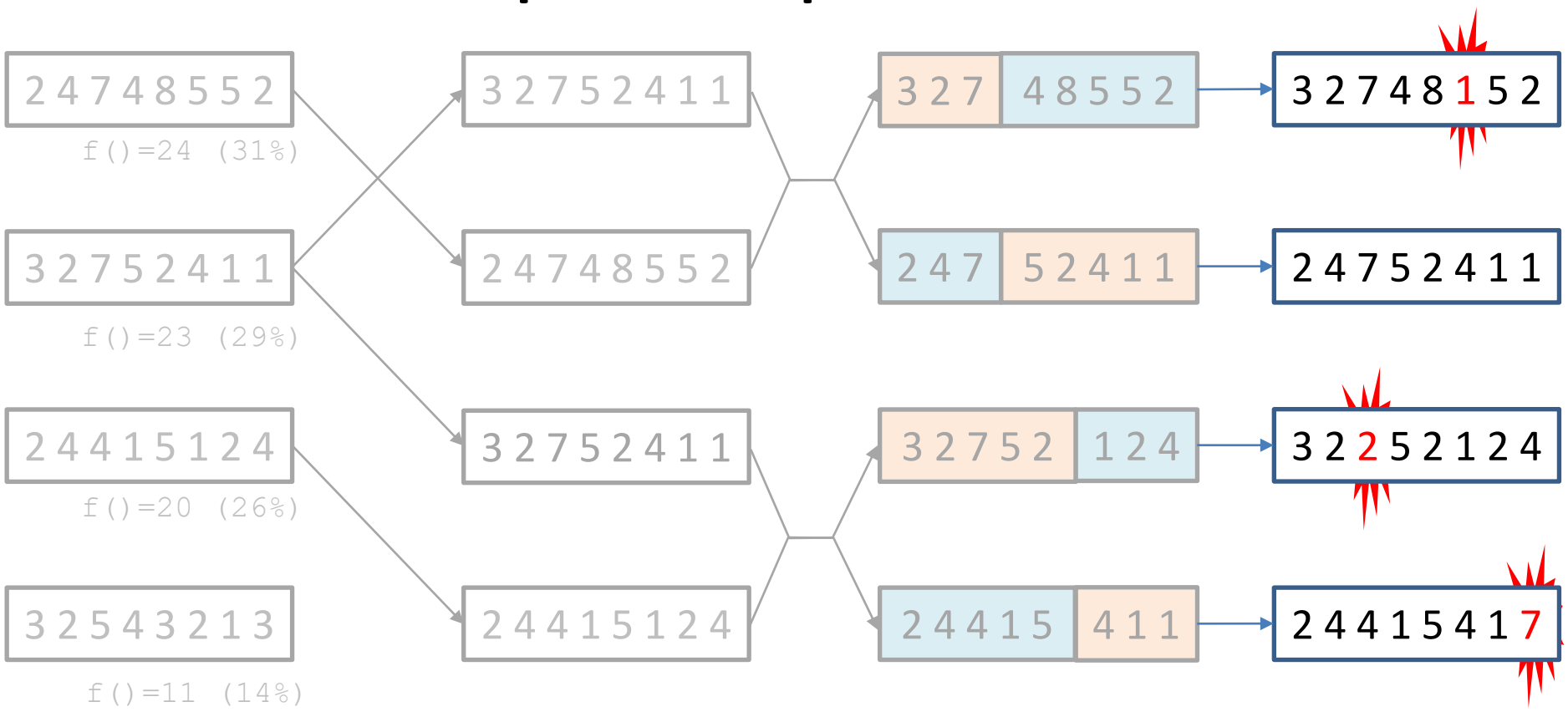

5
6
V=105; p=0.2
T=100
2
V=102; p=0.2
0
V=99; p= 20% * exp(Δ8/T)
p= 20% * exp(−1/100) = 0.198
8
1
V=97; 20% * exp(Δ1/T)
p = 20% * exp(−3/100) = 0.194
3
V=95; 25% * exp(Δ3/T)
p = 20% * exp(−5/100) = 0.19
4


5
6
V=105; p=0.2
T=0.01
2
V=102; p=0.2
0
V=99; p= 20% * exp(Δ8/T)
p= 20% * exp(−1/0.01) ≈ 0
8
1
V=97; 20% * exp(Δ1/T)
p = 20% * exp(−3/0.01) ≈ 0
3
V=95; 25% * exp(Δ3/T)
p = 20% * exp(−5/0.01) = ≈ 0
4

Genetic Algorithm

- **Population**
  - A set of candidate solutions/states
- **Crossover**
  - Combining parts of two states to create a child state
- **Mutation**
  - A random change in one bit of a state
- **Fitness function**
  - A numeric measure of the relative goodness of a state

8-queens problem



(a) Initial population

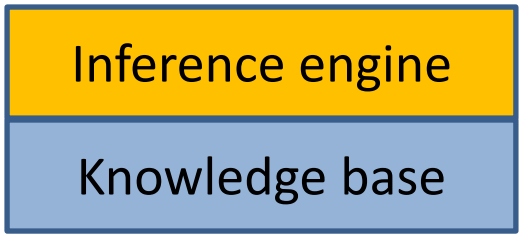
(b) Fitness function

(c) Selection

(d) Crossover

(e) Mutation

Declarative programming



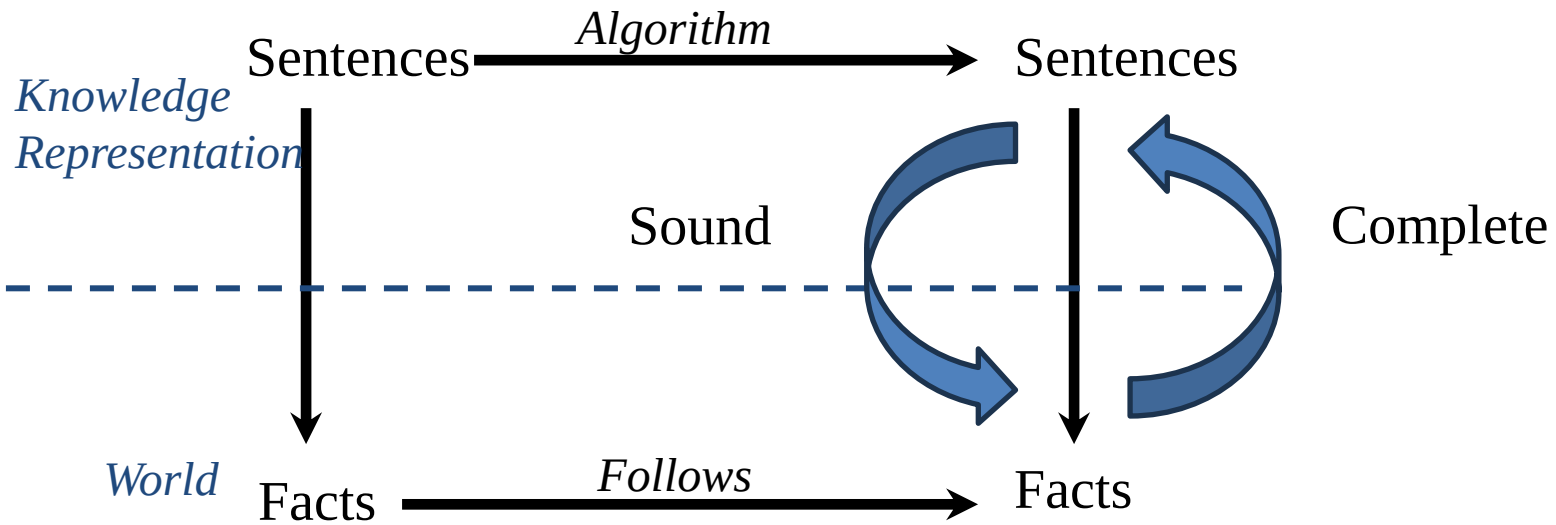
Domain-independent algorithms

Domain-specific content (facts)

- Knowledge base = set of sentences in a formal language
- Declarative approach to building an agent (or other system):
  - `Tell` it what it needs to know
- Then it can `Ask` itself what to do - answers should follow from the KB

Basic Idea of Logic

- By starting with true assumptions, you can deduce true conclusions.



Propositional logic

- Symbolic logic for manipulating propositions
  - Can be classified as either TRUE or FALSE
- **Logical constants**: true, false
- **Propositional symbols**: P, Q, S, ... (**atomic sentences**)
- Wrapping **parentheses**: (…)
- Sentences are combined by **connectives**:

∧ ...and	[conjunction]
∨ ...or	[disjunction]
→...implies	[implication / conditional]
↔..is equivalent	[biconditional]
~ ...not	[negation]

- **Literal**: atomic sentence or negated atomic sentence

Proof methods

- Proof methods divide into (roughly) two kinds:
- Application of inference rules
  - Legitimate (sound) generation of new sentences from old
  - Proof = a sequence of inference rule applications
    Can use inference rules as operators in a standard search algorithm
  - Typically require transformation of sentences into a normal form
- Model checking
  - truth table enumeration
  - improved backtracking, e.g., Davis--Putnam-Logemann-Loveland (DPLL)
  - heuristic search in model space (sound but incomplete)
    e.g., hill-climbing algorithms

“Proofs” using a truth table

- Premises (KB)

  1) Q

  2) $(P \wedge Q) \rightarrow R$

  3) $Q \rightarrow P$

- Does R follow?

A	B	$A \rightarrow B$
T	T	T
T	F	F
F	T	T
F	F	T

P	Q *	R	$(P \wedge Q) \rightarrow R$ *	$Q \rightarrow P$ *	R
True	T	T	T	T	T
True	T	F	F	T	F
T	F	T	T	T	T
T	F	F	T	T	F
F	T	T	T	F	T
F	T	F	T	F	F
F	F	T	T	T	T
F	F	F	T	T	F

“Proofs” using inference

RULE	PREMISE	CONCLUSION
Modus Ponens	$A, A \rightarrow B$	B
AND Introduction	A, B	$A \wedge B$
AND Elimination	$A \wedge B$	A
Double Negation	$\sim\sim A$	A
Resolution	$A \vee B, \sim B \vee C$	$A \vee C$
Unit resolution	$A \vee B, \sim B$	A

- Each can be shown to be sound/correct using a truth table

Example of a proof

- Premises

  1) Q

  2) $(P \wedge Q) \rightarrow R$

  3) $Q \rightarrow P$

- How to prove?

  – R

Example of a proof

1. Q
   - Premise 1
2. $Q \rightarrow P$
   - Premise 3
3. P
   - Modus ponens on 1 and 2
4. $(P \wedge Q) \rightarrow R$
   - Premise 2
5. $(P \wedge Q)$
   - AND introduction on 1 and 3
6. R
   - Modus ponens on 4 and 5

Proof by Resolution Refutation

- Does Premise (KB) $\rightarrow$ Conclusion (α)?

1. Convert all premise sentences (KB) to CNF
2. Add the *negated* conclusion
3. Repeatedly apply rule of resolution until
   - Derive FALSE (contradiction): Conclusion is **valid**
   - Can't apply any more: Conclusion **cannot be proved**

- Proof by contradiction

Example 1

- Premise:

  $P \vee Q$

  $P \rightarrow R$

  $Q \rightarrow R$

- Prove:

  R

1. $P \vee Q$
2. $\sim P \vee R$
3. $\sim Q \vee R$
4. $\sim R$
5. $Q \vee R$
   1. Resolution on 1,2
6. R
   1. Resolution on 3,5
7. FALSE
   1. Resolution on 4,6