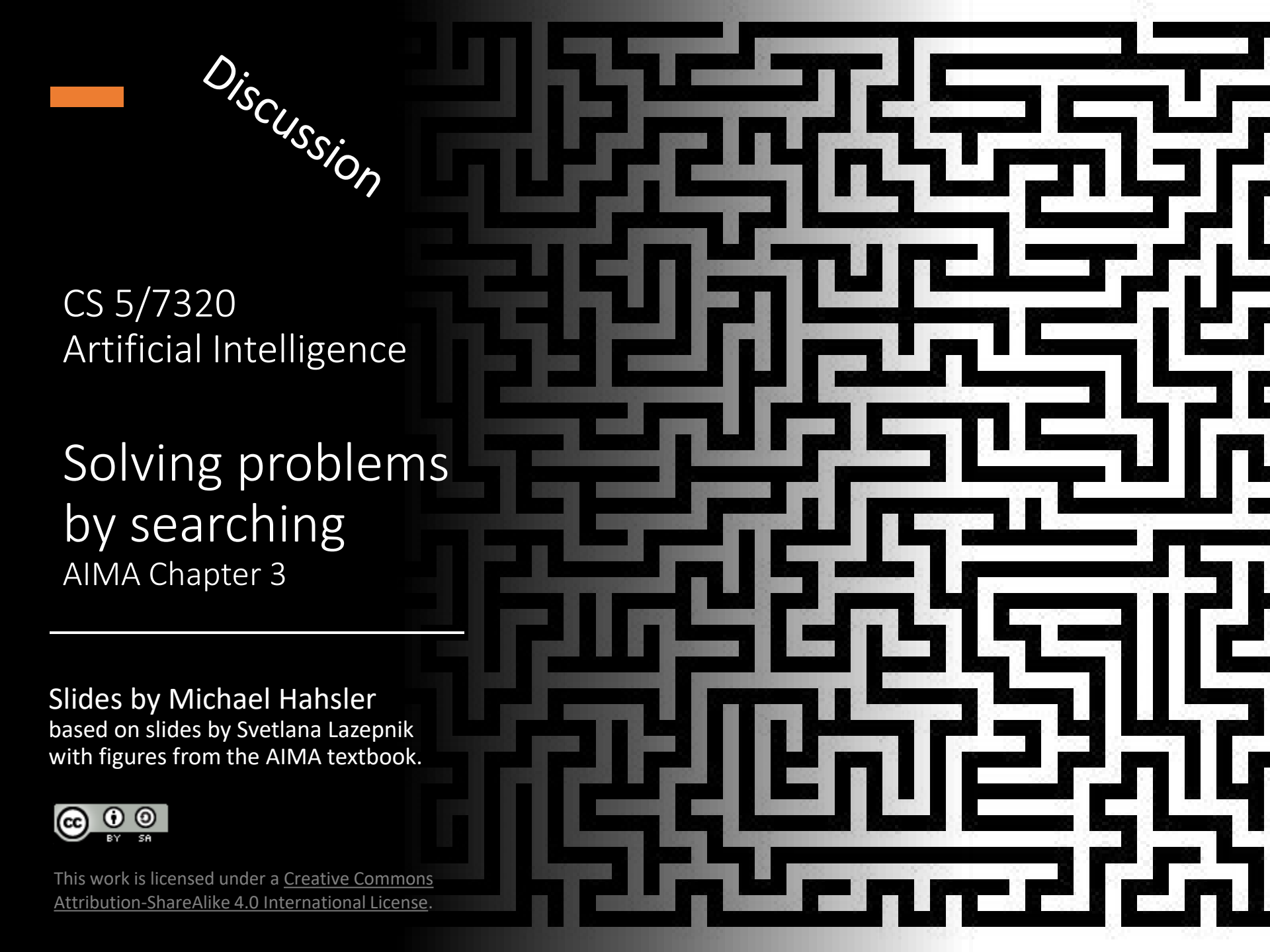


CS 5/7320
Artificial Intelligence

Solving problems by searching

AIMA Chapter 3

Slides by Michael Hahsler
based on slides by Svetlana Lazepnik
with figures from the AIMA textbook.

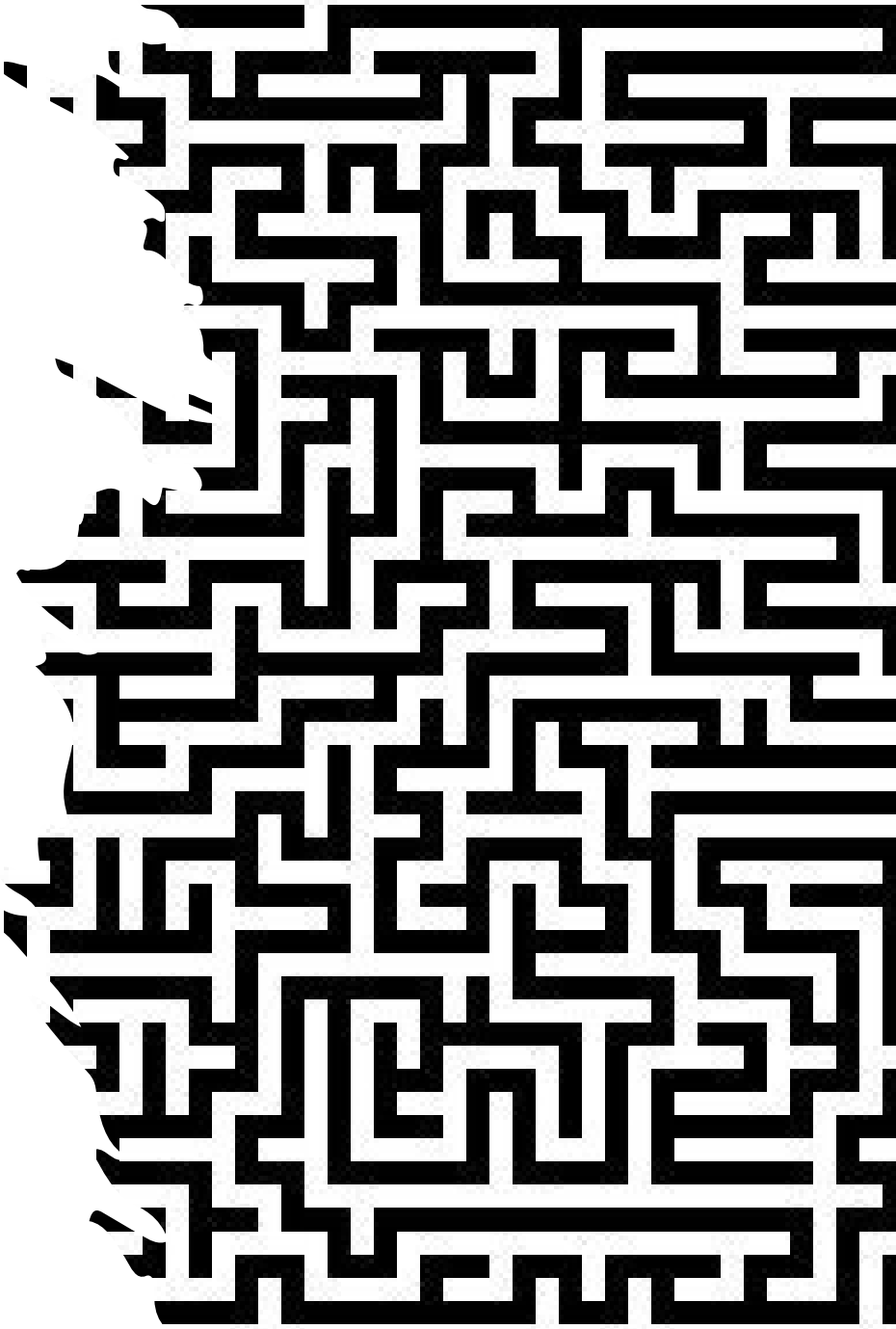
Assignment 1

Discuss:

- Robustness (Obstacles)
- Noisy sensors

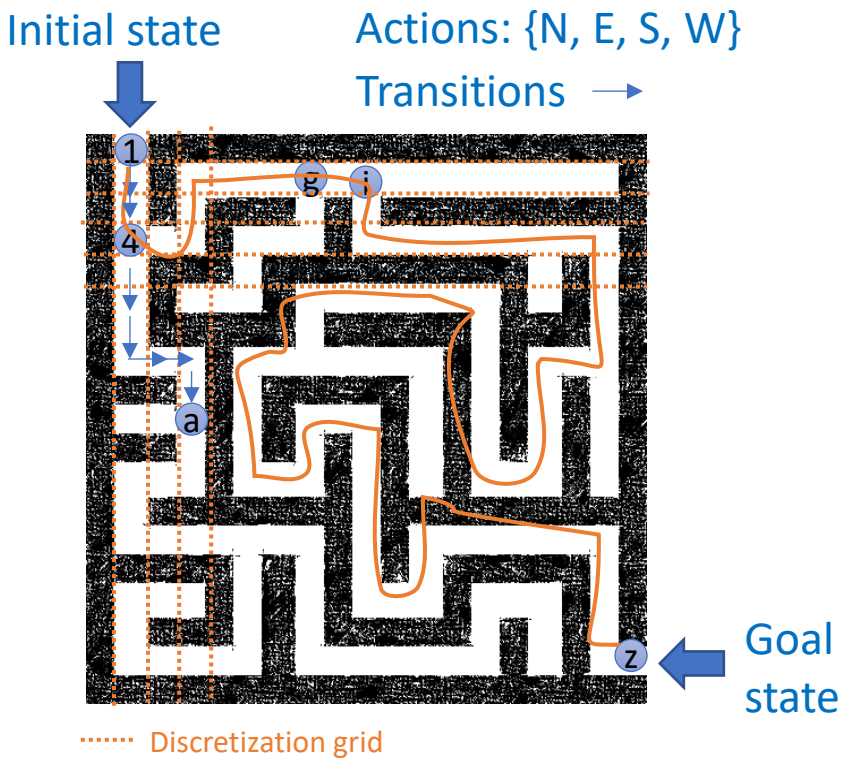
Module Review

Intro and Uninformed Search



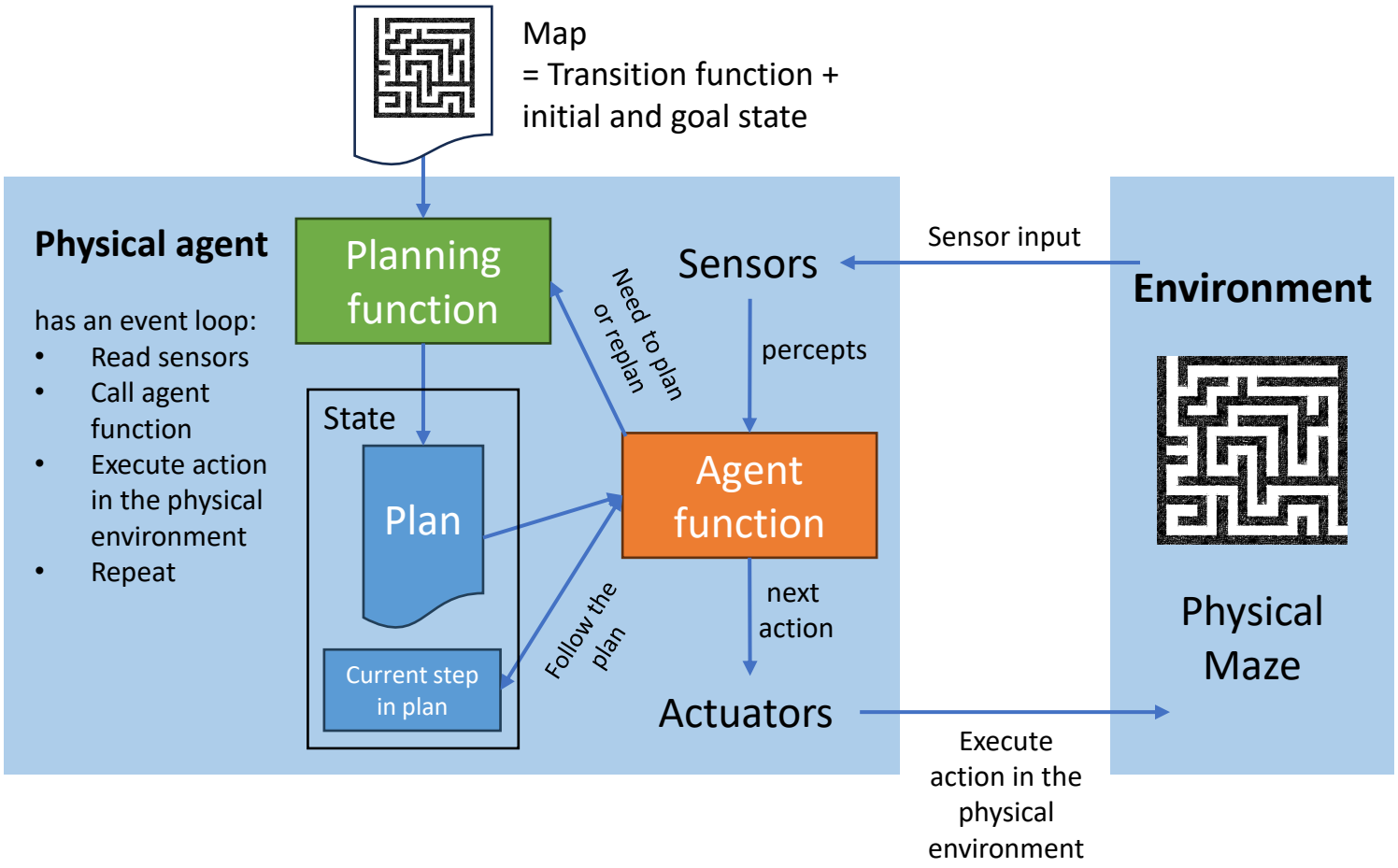
Definition of a Search Problem

- **Initial state:** state description
- **Actions:** set of possible actions A
- **Transition model:** a function that defines the new state resulting from performing an action in the current state
- **Goal state:** state description
- **Path cost:** the sum of *step costs*



Important: The **state space** is typically too large to be enumerated, or it is continuous. Therefore, the problem is defined by initial state, actions and the transition model and not the set of all possible states.

Complete Planning Agent to Solve a Maze



- The event loop calls the agent function for the next action.
- The agent function follows the plan or calls the planning function if there is no plan yet or it thinks the current plan does not work based on the percepts (replanning).

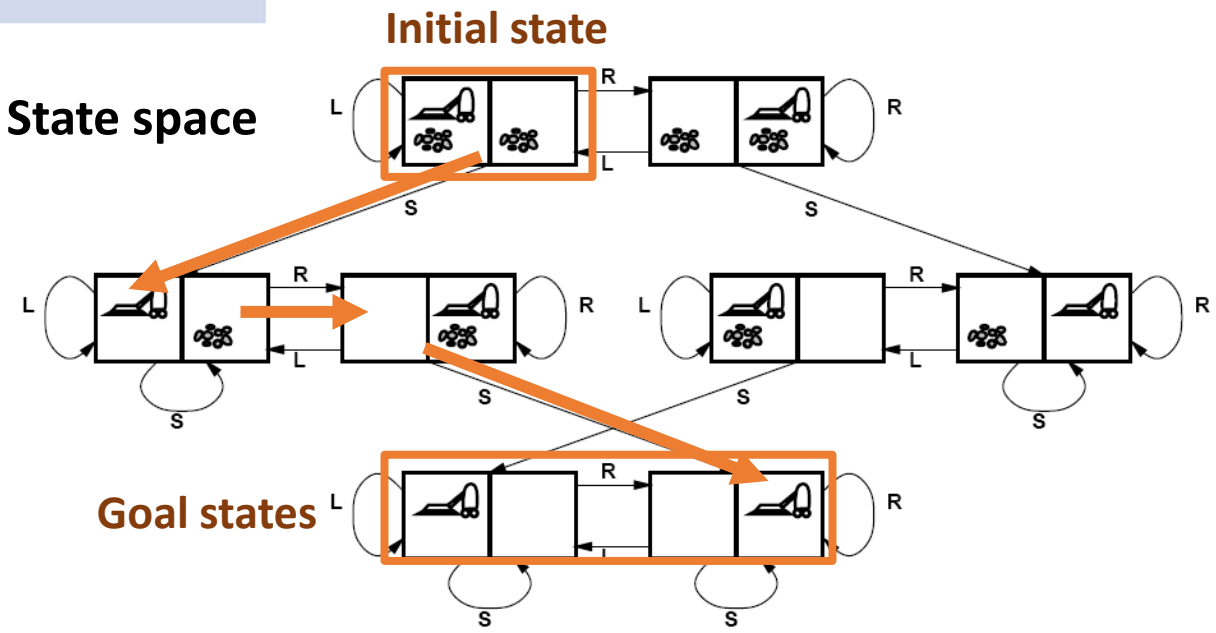
Solving Search Problems

Given a search problem definition

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

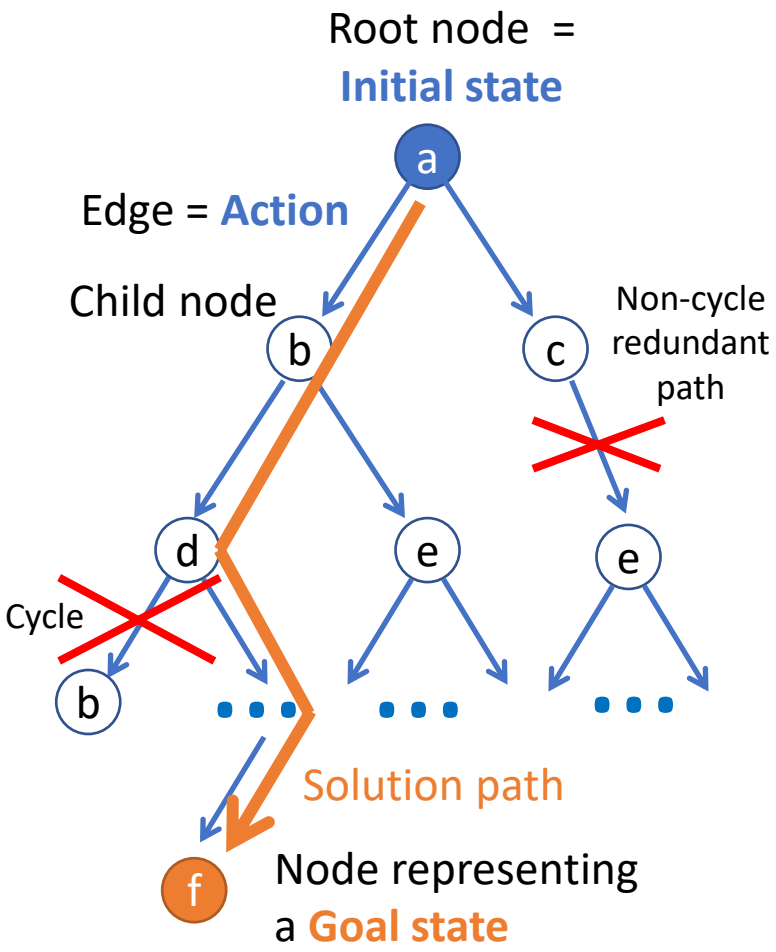
How do we find the optimal solution (sequence of actions/states)?

Construct a search tree for the state space graph!



Creating a Search Tree

- Superimpose a "what if" tree of possible actions and outcomes (states) on the state space graph.
- The **Root node** represents the initial stare.
- An action child node is reached by an **edge** representing an action. The corresponding state is defined by the transition model.
- Trees cannot have **cycles (loops).** Cycles in the search space must be broken to prevent infinite loops.
- Trees cannot have **multiple paths to the same state.** These are called redundant paths. Removing other redundant paths improves search efficiency.
- A **path** through the tree corresponds to a sequence of actions (states).
- A **solution** is a path ending in a node representing a goal state.
- **Nodes vs. states:** Each tree node represents a state of the system. If redundant path cannot be prevented then state can be represented by multiple nodes in the tree.



Differences Between Typical Tree Search and AI Search

Typical tree search

- Assumes a given tree that fits in memory.

- Trees have by construction no cycles or redundant paths.

AI tree/graph search

- The search tree is too large to fit into **memory**.
  a. **Builds parts of the tree** from the initial state using the transition function representing the graph.
  b. **Memory management** is very important.

- The search space is typically a very large and complicated graph. Memory-efficient **cycle checking** is very important to avoid infinite loops or minimize searching parts of the search space multiple times.
- Checking redundant paths often requires too much memory and we accept searching the same part multiple times.

Summary:
All Search Strategies

b: maximum branching factor of the search tree
d: depth of the optimal solution
m: maximum length of any path in the state space
C*: cost of optimal solution

	Algorithm	Complete?	Optimal?	Time complexity	Space complexity
Uninformed Search	**BFS (Breadth-first search)**	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
	Uniform-cost Search	Yes	Yes	Number of nodes with $g(n) \leq C^*$	
	DFS	In finite spaces (cycles checking)	**No**	$O(b^m)$	$O(bm)$
	IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$
Informed Search	**Greedy best-first Search**	In finite spaces (cycles checking)	No	Depends on heuristic Best case: $O(bd)$ Worst case: $O(b^m)$	
	A* Search	Yes	Yes	Number of nodes with $g(n) + h(n) \leq C^*$	

With a good heuristic

Breadth-First Search (BFS)

All nodes are in memory!

Expansion rule: Expand shallowest unexpanded node in the frontier (=**FIFO**).

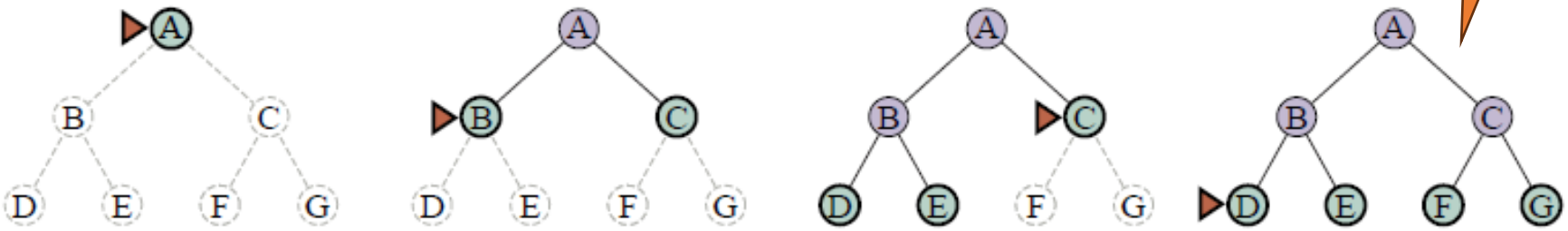


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Data Structures

- **Frontier** data structure: holds references to the green nodes (green) and is implemented as a FIFO **queue**.
- **Reached** data structure: holds references to all visited nodes (gray and green) and is used to prevent visiting nodes more than once (cycle and redundant path checking).
- Builds a **complete tree** with links between parent and child.

This is a generalization of Breadth-first search that expands the search based on cost and not on the number of steps.

Implementation: Best-First Search Strategy

```
function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)

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)
```

The order for expanding the frontier is determined by *f(n)* = path cost from the initial state to node *n*.

This check is added to BFS! It visits a node again if it can be reached by a better (cheaper) path.

Depth-First Search (DFS)

- **Expansion rule**: Expand deepest unexpanded node in the frontier (last added).
- **Frontier**: **stack** (LIFO)
- **No reached data structure!**

  Cycle checking checks only the current path.

  Redundant paths can not be identified and lead to replicated work.

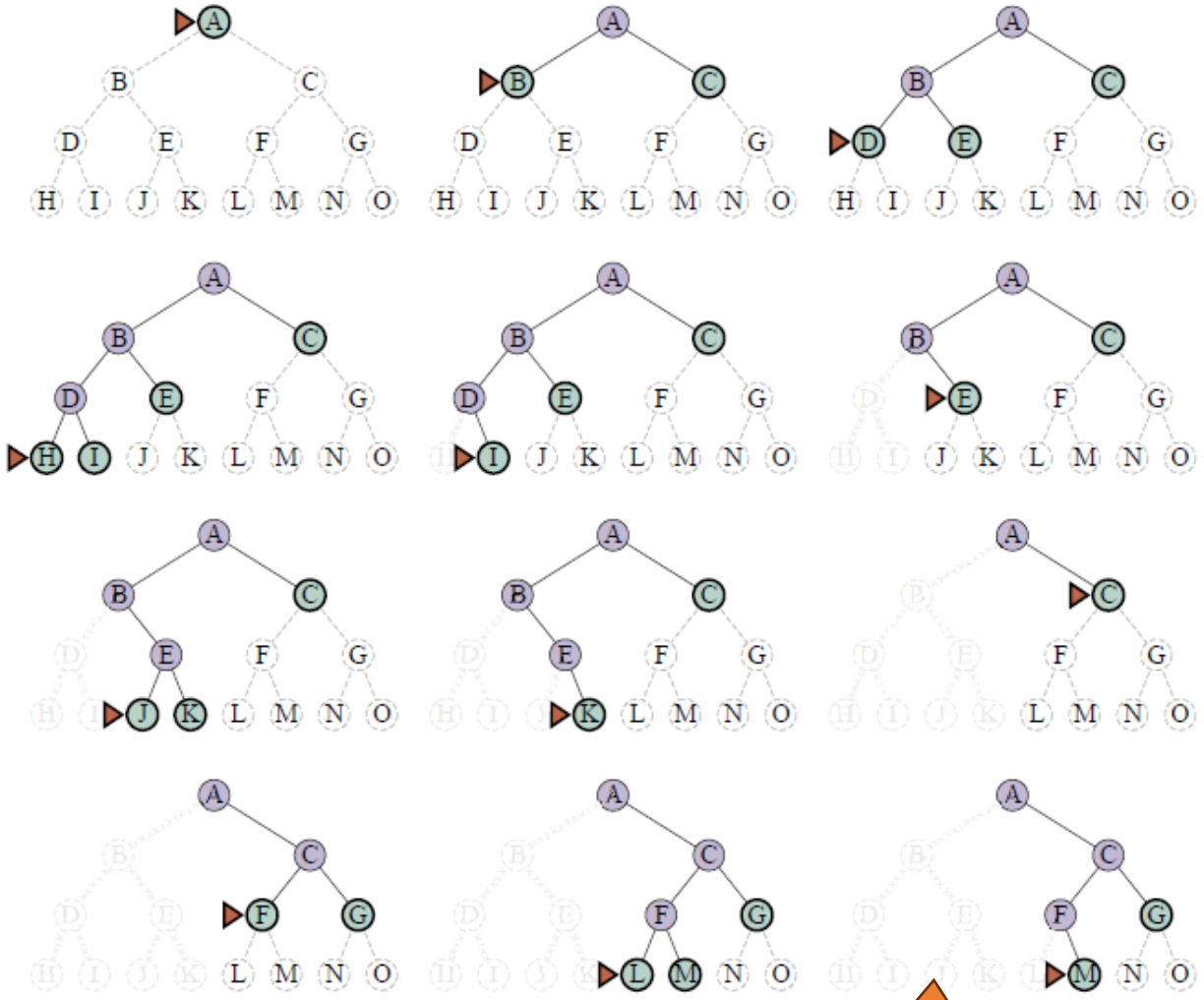
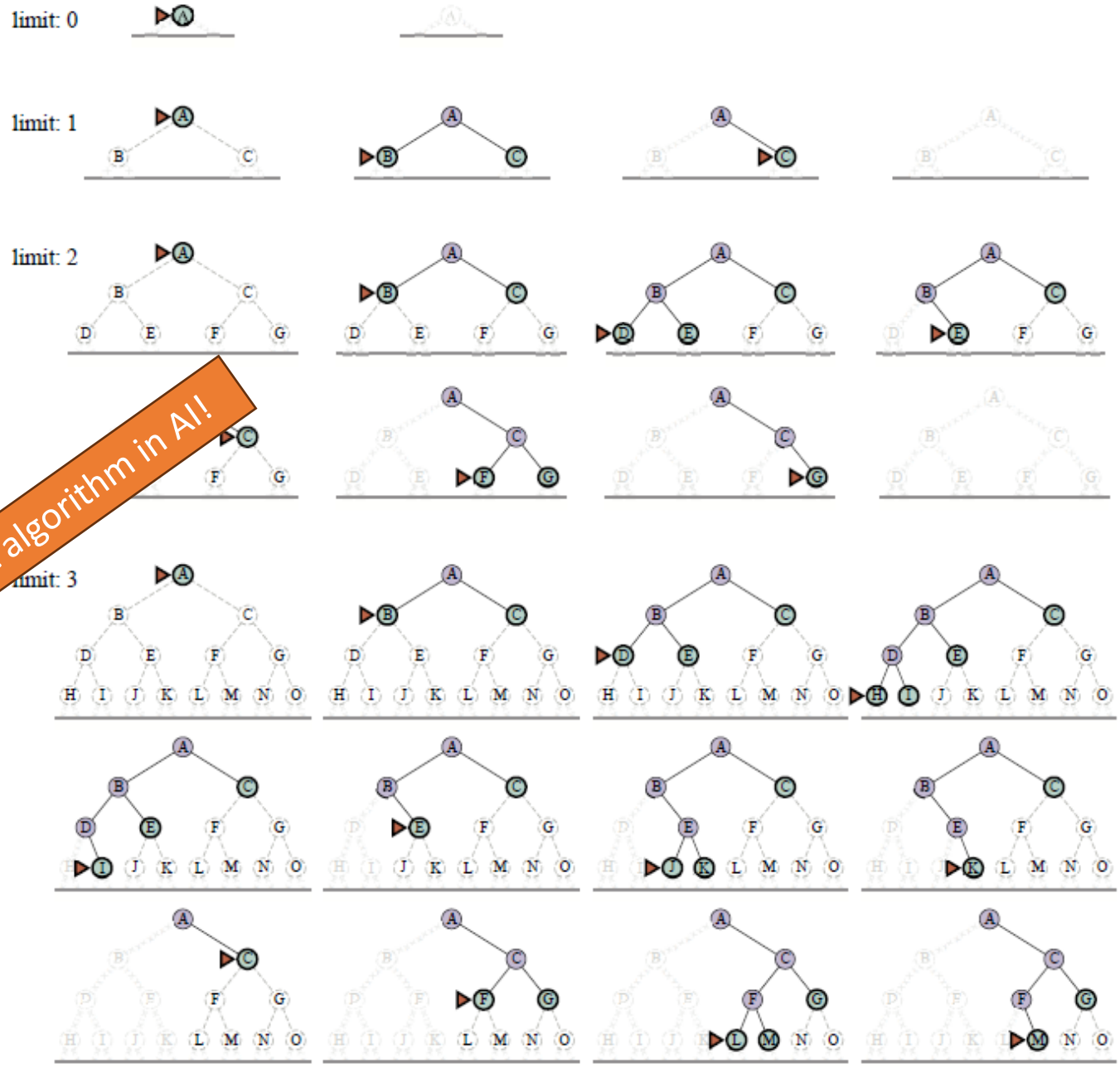


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of depth-first search on a binary tree from start state A to goal M. The front[ier] is a single marking the node to be expanded next. Previously expanded nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

Memory management: only the current path is in memory!

Iterative Deepening Search (IDS)



This is a very important algorithm in AI!

DFS is implemented very differently to BFS!

Implementation: DFS

- DFS could be implemented like BFS/Best-first search, just taking the last element from the frontier (LIFO). However, to reduce the space complexity to $O(bm)$, **no reached data structure can be used!**
- Options:
  - **Iterative implementation**: Build the tree, and abandoned branches are removed from memory. Cycle checking is only done against the current path. This is similar to Backtracking search.
  - Recursive implementation: Cycle checking is an issue because the current path is stored in the function call stack, which is not accessible to the function. An additional data structure that contains the nodes in the current path can be used.

DFS uses $\ell = \infty$

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Memory management: remove nodes for abandoned branches here!

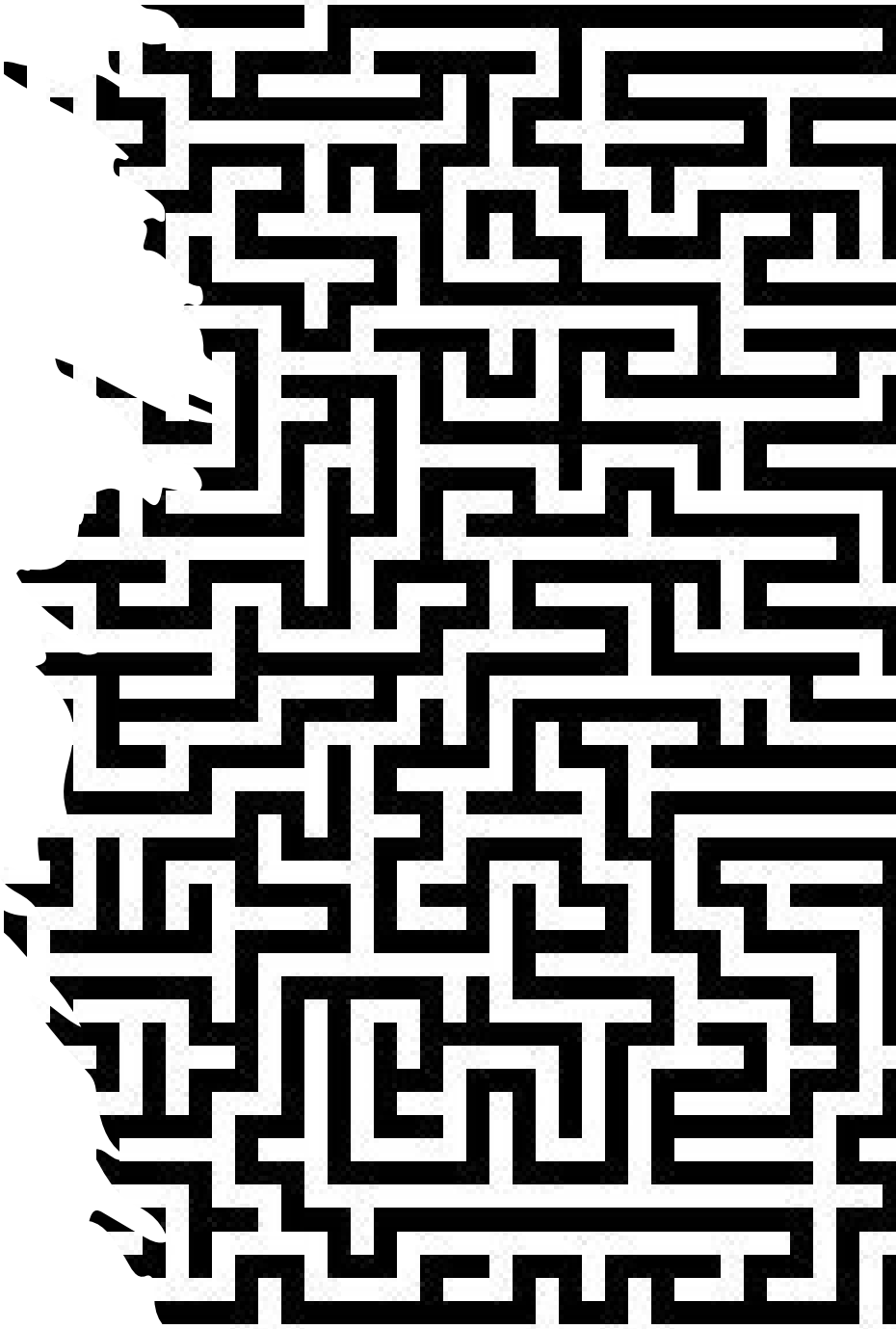
Cycles: Prevent cycles by checking against the current path. We also need to ensure that the frontier does not contain the same state more than once.

Redundant paths: We cannot prevent other redundant paths.

See BFS for function EXPAND.

Module Review

Statespace and Search Complexity



State Space vs. Search Tree Size

- Space and time complexity depend on the **number of tree nodes** searched (created and visited) till a goal node is found. For a tree with n nodes we have:

$$O(n)$$

- **Remember**: For perfect cycle checking and redundant path elimination, we have a 1:1 mapping between nodes and states:

$$\text{Nodes in the search tree} = \text{states in the search space}$$

  Otherwise, we may have multiple nodes representing a state.

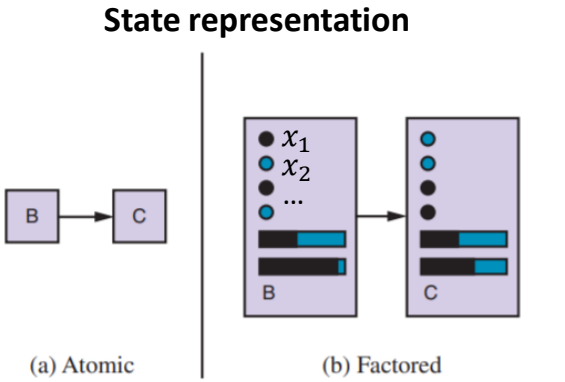
- We have the following options to estimate n for a search problem:
  a. **Estimate the reachable state space size.**
  b. **Estimate the number of searched tree nodes.**

- Estimating the complexity is important to judge:
  - How difficult is the problem?
  - What algorithm will fit in memory?
  - Can we find a solution fast enough?
  - Can we find the optimal solution, or do we need to use a heuristic?

State Space Size Estimation

State Space

- Number of different states the agent and environment can be in.
- **Reachable states** are defined by the initial state and the transition model. Only reachable states are important for search.



The factored state consists of variables called fluents that represent conditions that can change over time.

Estimation

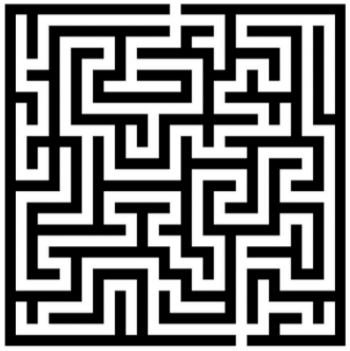
- Even if the used algorithm represents the state space using atomic states, we may know the internal (factored) representation. It can be used to estimate the problem size.
- The basic rule is to estimate the state space size for factored state representation with l fluents (variables) as:

$$n = |X_1| \times |X_2| \times \cdots \times |X_l|$$

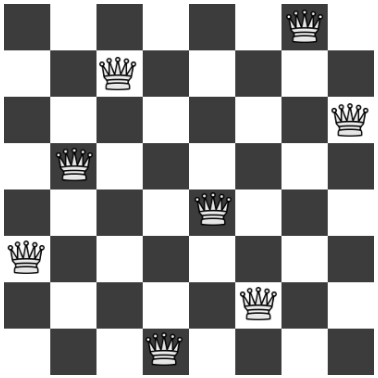
where $|\cdot|$ is the number of possible values.

Examples: What is the State Space Size?

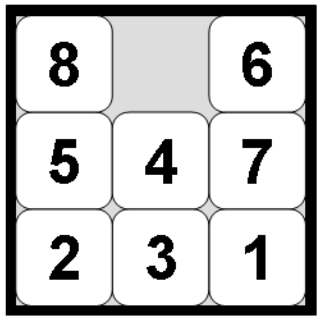
Often a rough upper limit is sufficient to determine how hard the search problem is.



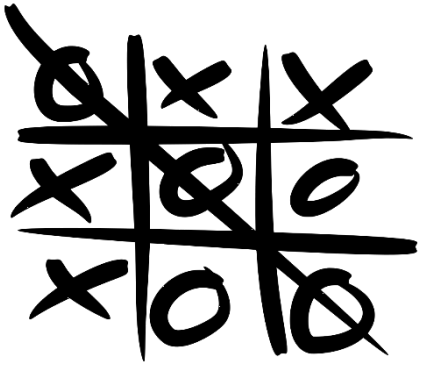
Maze



8-queens problem



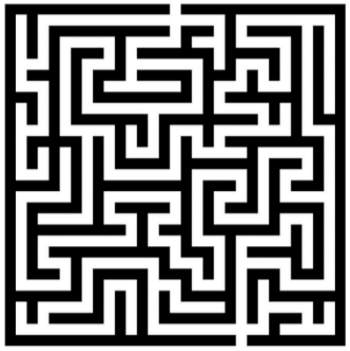
8-puzzle problem



Tic-tac-toe

Examples: What is the State Space Size?

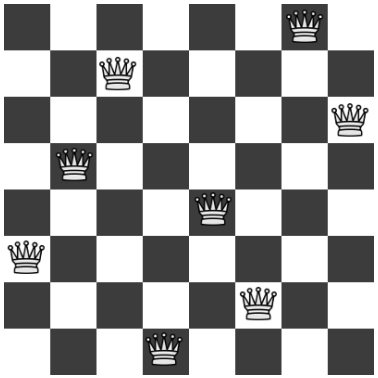
Often a rough upper limit is sufficient to determine how hard the search problem is.



Maze

Positions the agent can be in.

n = Number of white squares.



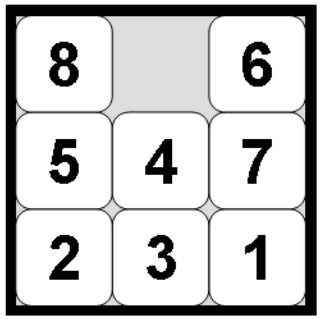
8-queens problem

All arrangements with 8 queens on the board.

$n < 2^{64} \approx 1.8 \times 10^{19}$

We can only have 8 queens:

$n = \binom{64}{8} \approx 4.4 \times 10^{9}$



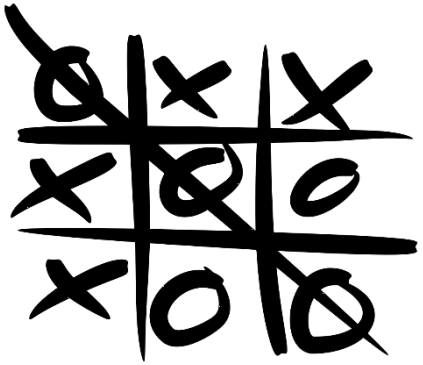
8-puzzle problem

All arrangements of 9 elements.

$n \leq 9!$

Half is unreachable:

$n = \frac{9!}{2} = 181{,}440$



Tic-tac-toe

All possible boards.

$n < 3^9 = 19{,}683$

Many boards are not legal (e.g., all x's)

The actual number can be obtained by a depth-first traversal of the game tree.

Example: What determines the Search Tree Size?

- b: maximum branching factor = number of available actions?

  3

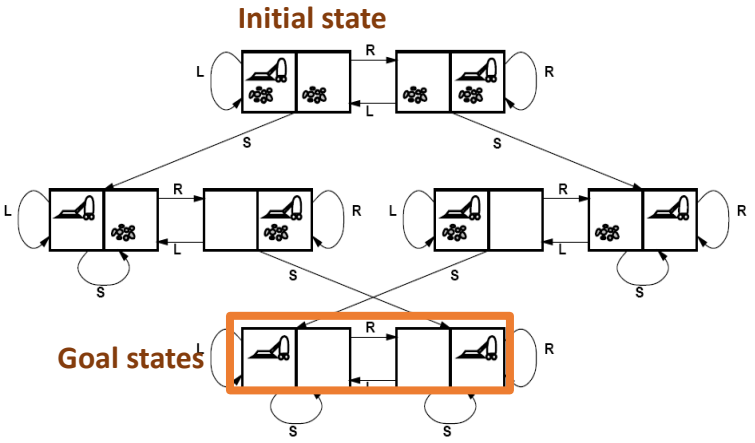
- m: the number of actions in any path? Without loops!

  4

- d: depth of the optimal solution?

  3

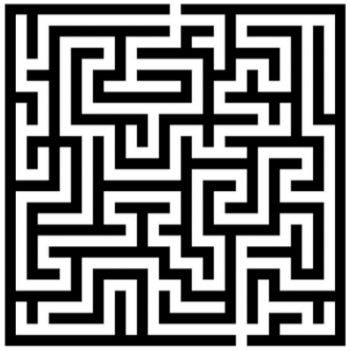
State Space with Transition Model



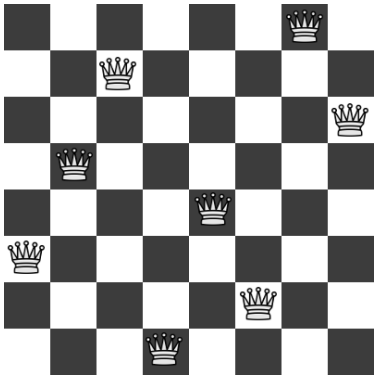
Examples: What determines the Search Tree Size?

b: maximum branching factor
m: max. depth of tree
d: depth of the optimal solution

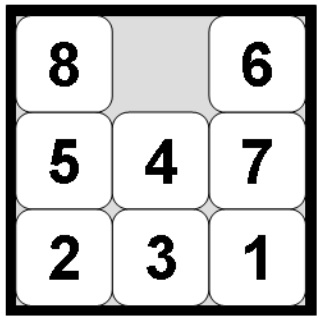
Often a rough upper limit is sufficient to determine how hard the search problem is.



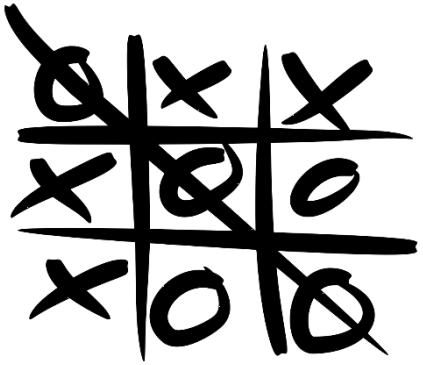
Maze



8-queens problem



8-puzzle problem



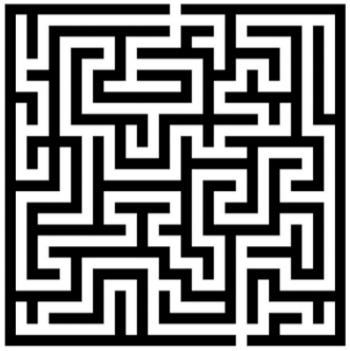
Tic-tac-toe

$b =$
$m =$
$d =$

Examples: What determines the Search Tree Size?

b: maximum branching factor
m: max. depth of tree
d: depth of the optimal solution

Often a rough upper limit is sufficient to determine how hard the search problem is.

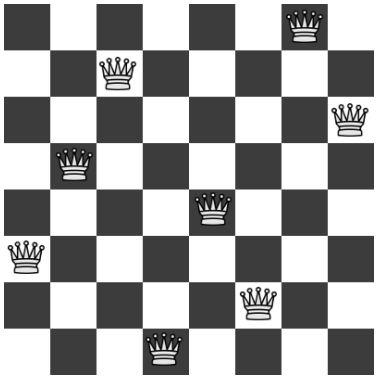


Maze

$b = 4$ actions

$m =$ longest path to the goal or a dead end (bounded by $x \times y$)

$d =$ shortest path to the goal (bounded by $x \times y$)

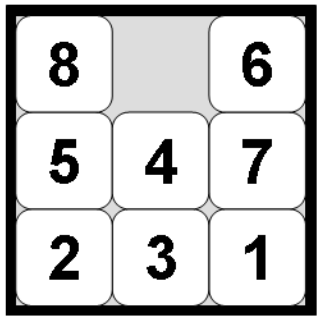


8-queens problem

$b =$? What are the actions? Moving one Queen: $64 - 7 = 57$

$m =$ We may have to try all: $\binom{64}{8} \approx 4.4 \times 10^9$

$d =$ move each queen in the right spot = 8

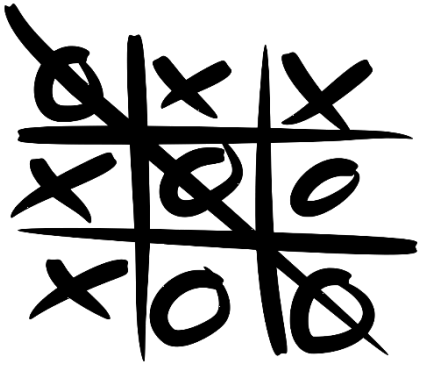


8-puzzle problem

$b = 4$ actions to move the empty tile.

$m =$ Try all $O(9!)$

$d =$???



Tic-tac-toe

$b = 9$ actions for the first move.
$m = 9$

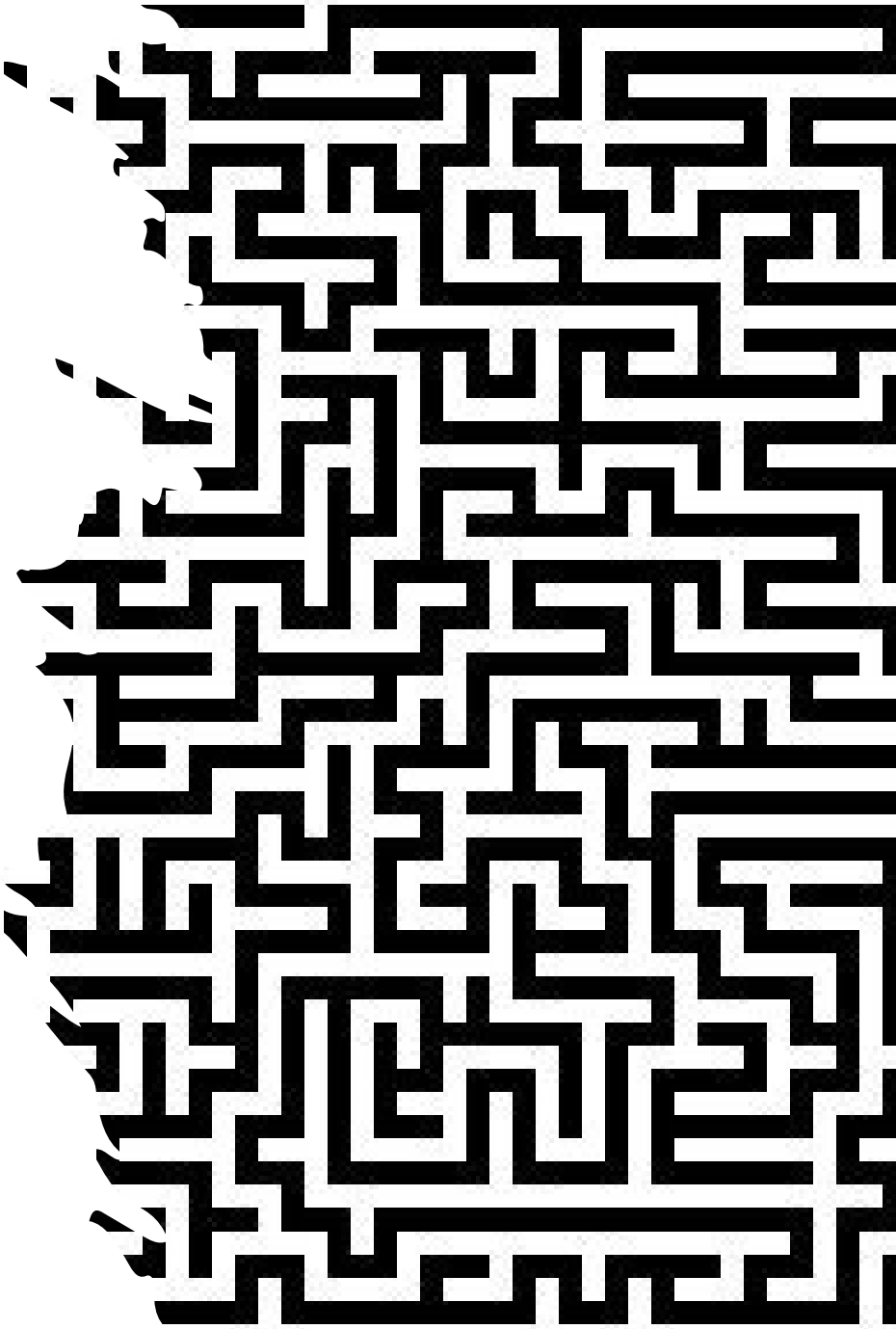
$d = 9$ (if both play optimal)

Assignment 2

Introduction

Module Review

Informed Search



Summary:
All Search Strategies

b: maximum branching factor of the search tree
d: depth of the optimal solution
m: maximum length of any path in the state space
C*: cost of optimal solution

	Algorithm	Complete?	Optimal?	Time complexity	Space complexity
Uninformed Search	**BFS (Breadth-first search)**	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
	Uniform-cost Search	Yes	Yes	Number of nodes with $g(n) \leq C^*$	
	DFS	In finite spaces (cycles checking)	**No**	$O(b^m)$	$O(bm)$
	IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$
Informed Search	**Greedy best-first Search**	In finite spaces (cycles checking)	No	Depends on heuristic Best case: $O(bd)$ Worst case: $O(b^m)$	
	A* Search	Yes	Yes	Number of nodes with $g(n) + h(n) \leq C^*$	

With a good heuristic

Properties of Heuristic Functions

- Evaluates a given node n.
- Provide a **good approximation** of the actual cost from node n to the goal state.
- Can be computed using additional **information that is known** to the agent or can be obtained via percepts.
- Are **fast to compute** (compared to solving the problem).

- For A* Search: be **admissible** (never overestimate the cost)

- Search algorithms will expand nodes with better heuristic values/estimated total cost first.

Implementation

Greedy Best-First search

Best-First Search + Expand the frontier using $f(n) = h(n)$

A* Search

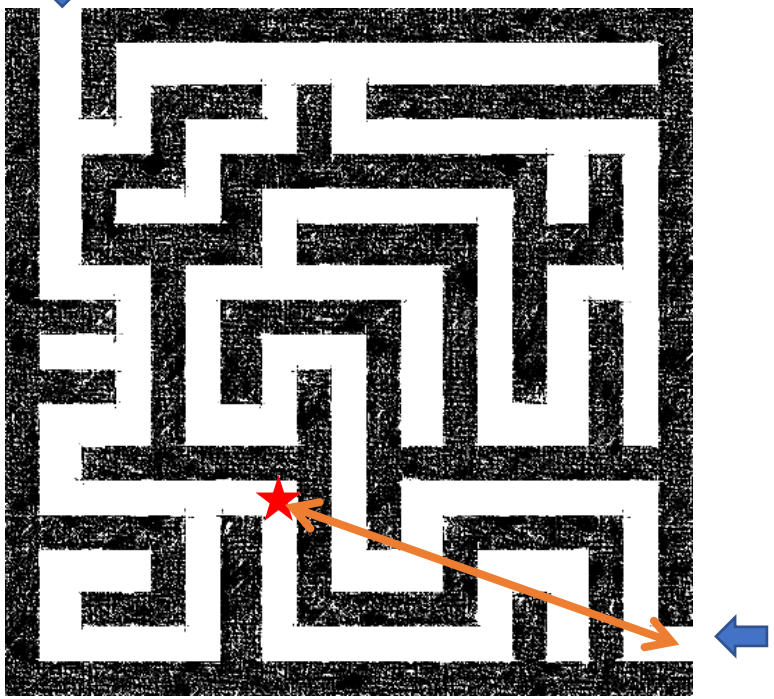
Best-First Search + Expand the frontier using $f(n) = h(n) + \boldsymbol{g(n)}$

Heuristics from Relaxed Problems

What relaxations are used in these two cases?

Euclidean distance

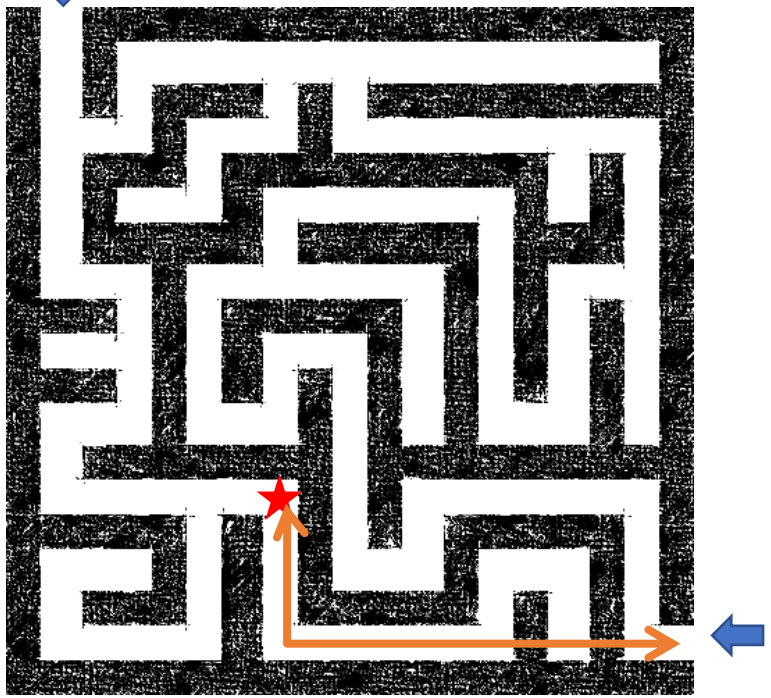
Start state



Goal state

Manhattan distance

Start state



Goal state

Which heuristic is admissible for A* search?

Satisficing Search: Weighted A* Search

- Often it is sufficient to find a **"good enough" solution** if it can be found very quickly or with way less computational resources. I.e., expanding fewer nodes.

- We could use inadmissible heuristics in A* search (e.g., by multiplying $h(n)$ with a factor W) that sometimes overestimate the optimal cost to the goal slightly.
  1. It potentially reduces the number of expanded nodes significantly.
  2. **This will break the algorithm's optimality guaranty!**

$$f(n) = g(n) + W \times h(n)$$

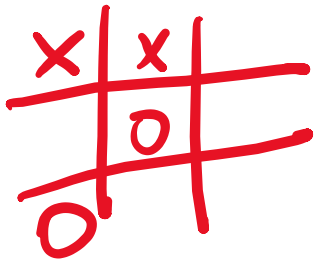
Weighted A* search:	$\boldsymbol{g(n) + W \times h(n)}$	$\boldsymbol{(1 < W < \infty)}$

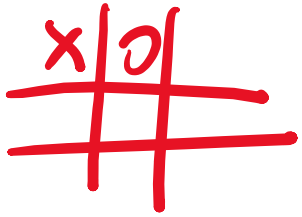
The presented algorithms are special cases:

A* search:	$g(n) + h(n)$	$(W = 1)$
Uniform cost search/BFS:	$g(n)$	$(W = 0)$
Greedy best-first search:	$h(n)$	$(W = \infty)$

Case Study: A Heuristic for Tic-Tac-Toe

- Define the goal states:
- What is the cost that needs to be estimated?
- What could be used as a heuristic value for these boards (assume you play x):





- How do you calculate the heuristic value?
- Is the heuristic admissible?
- Does the heuristic use a relaxation?

Assignment 2

Discussion