



INTRODUCTION TO ARTIFICIAL INTELLIGENCE

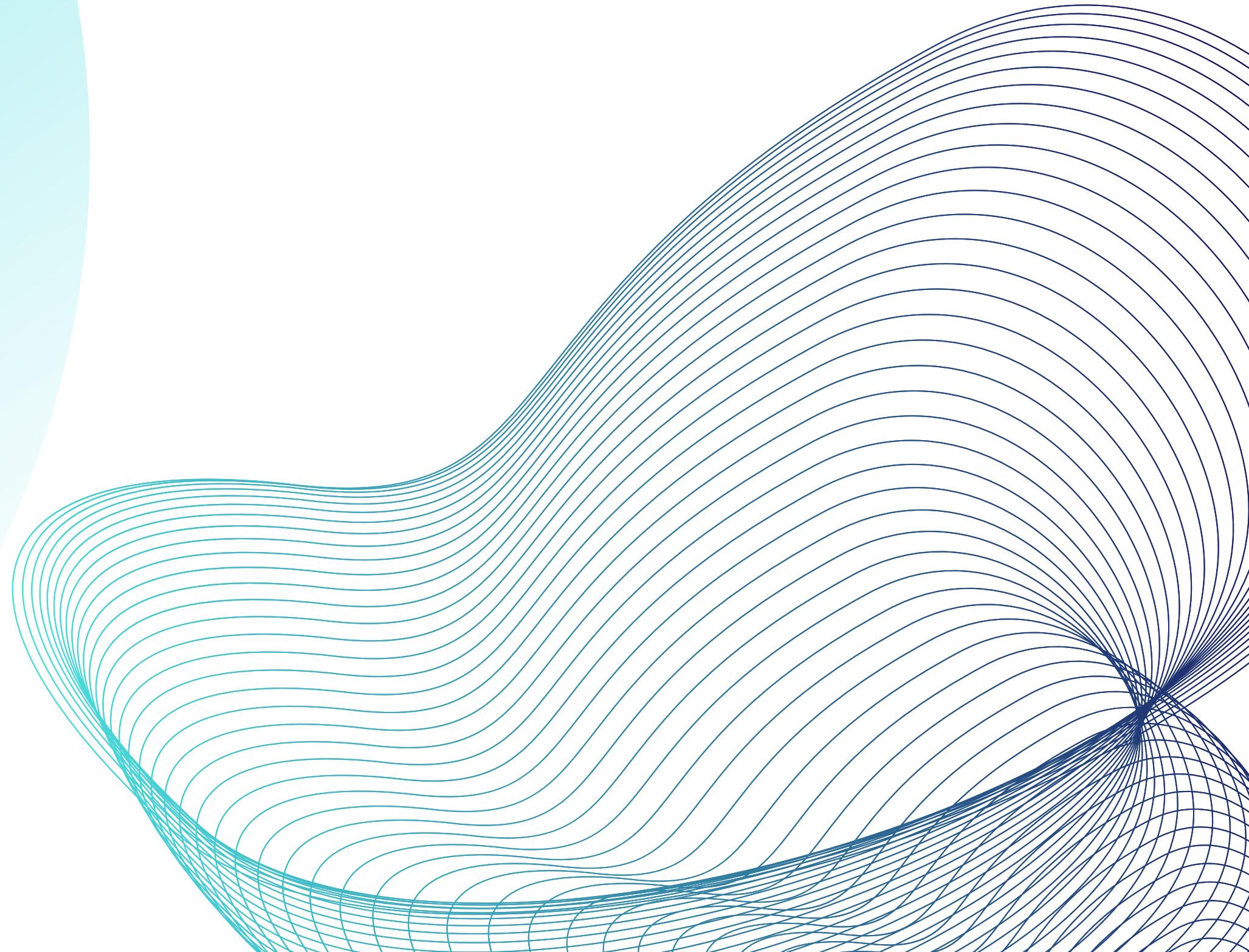
PGS. TS. Nguyễn Thanh Bình





SEARCH ALGORITHMS

- Uninformed search algorithms
- Informed search algorithms



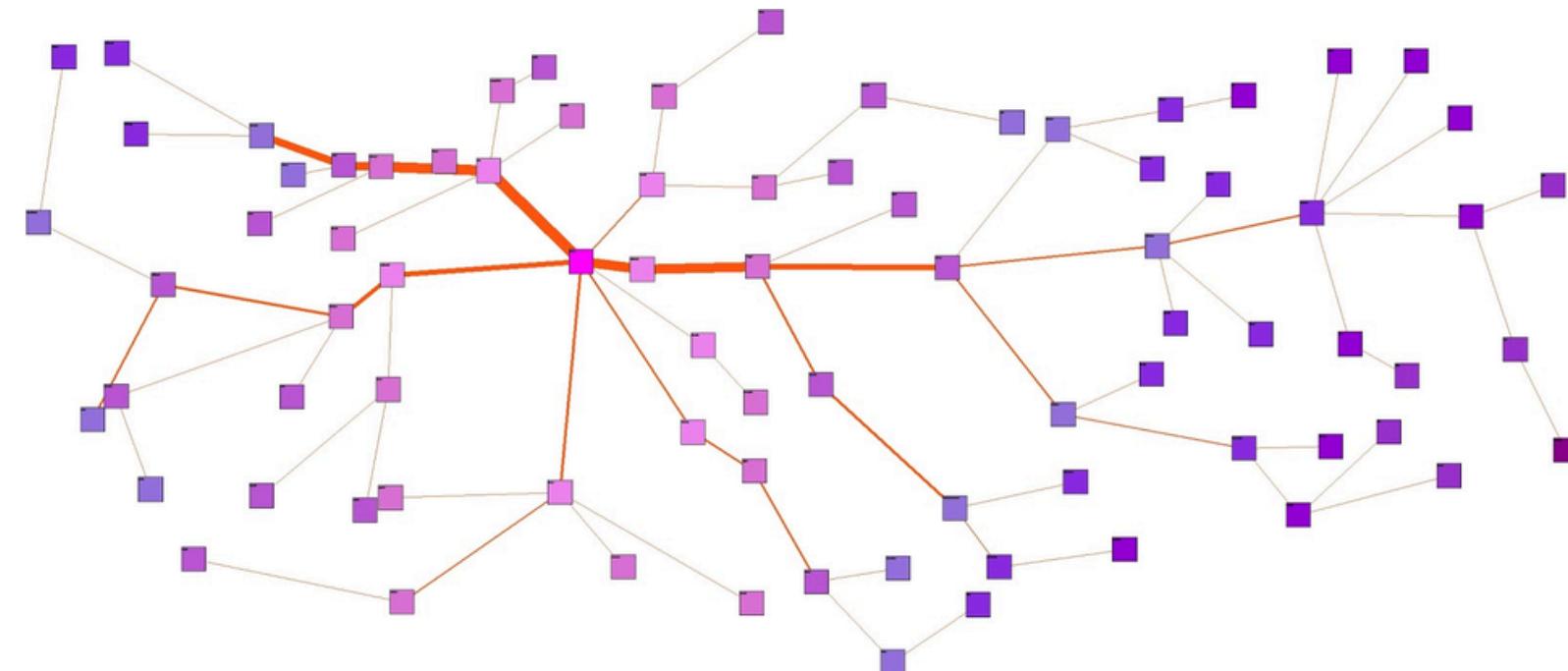
UNINFORMED SEARCH STRATEGIES

- Also called “blind search”.
- No additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from any non-goal state.
- Different from the informed strategies or heuristic strategies.

UNINFORMED SEARCH STRATEGIES

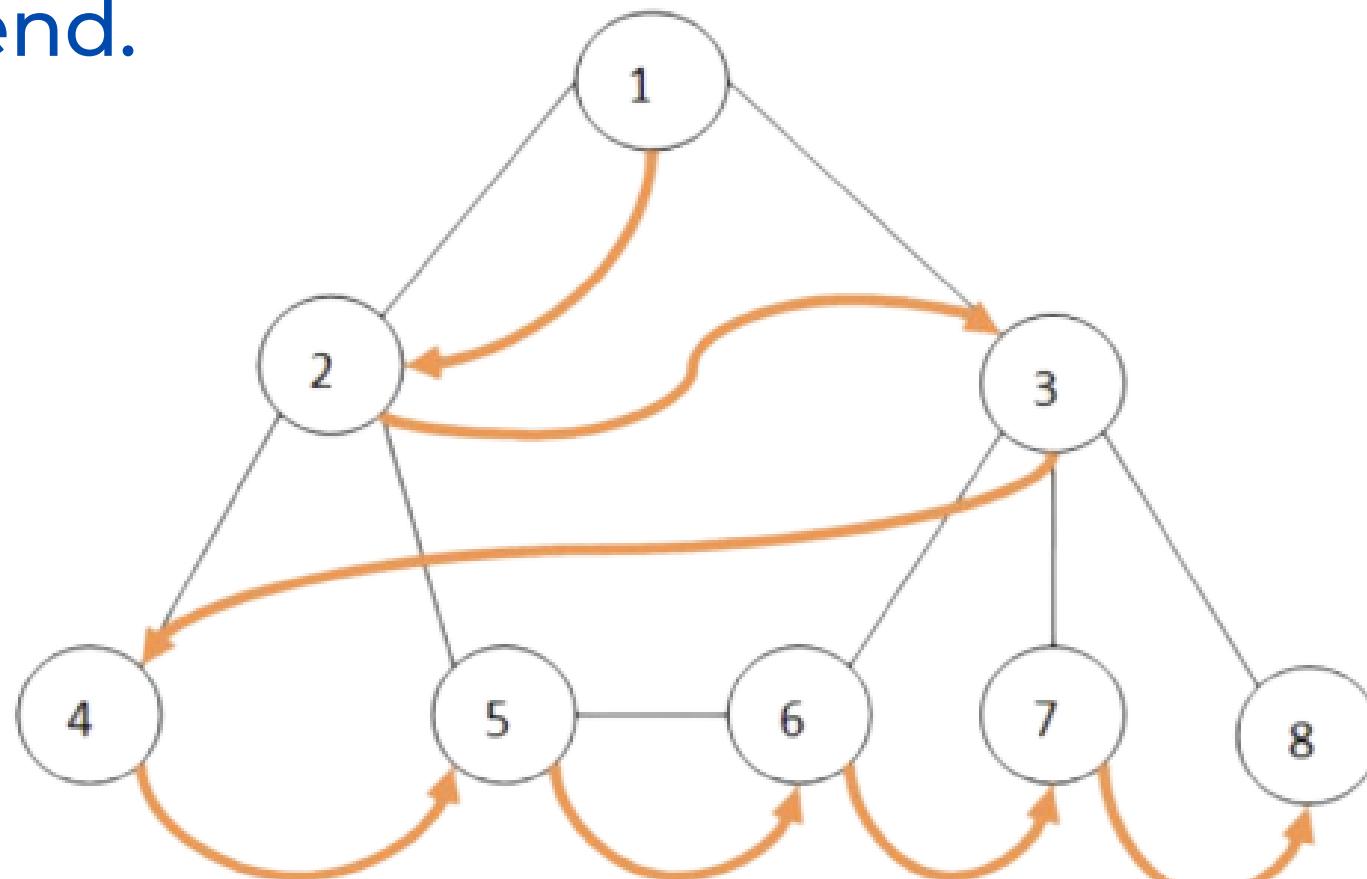
Several uninformed search strategies:

- Breadth-first search.
- Uniform-cost search.
- Depth-first search.
- Depth-limited search.
- Iterative deepening search



BREADTH-FIRST SEARCH

- Expand **shallowest** unexpanded node.
- Implementation: fringe is a **First-In-First-Out** queue, i.e., new successors go at end.



Complete?
Time?
Space?
Optimal?
→HOMEWORK!

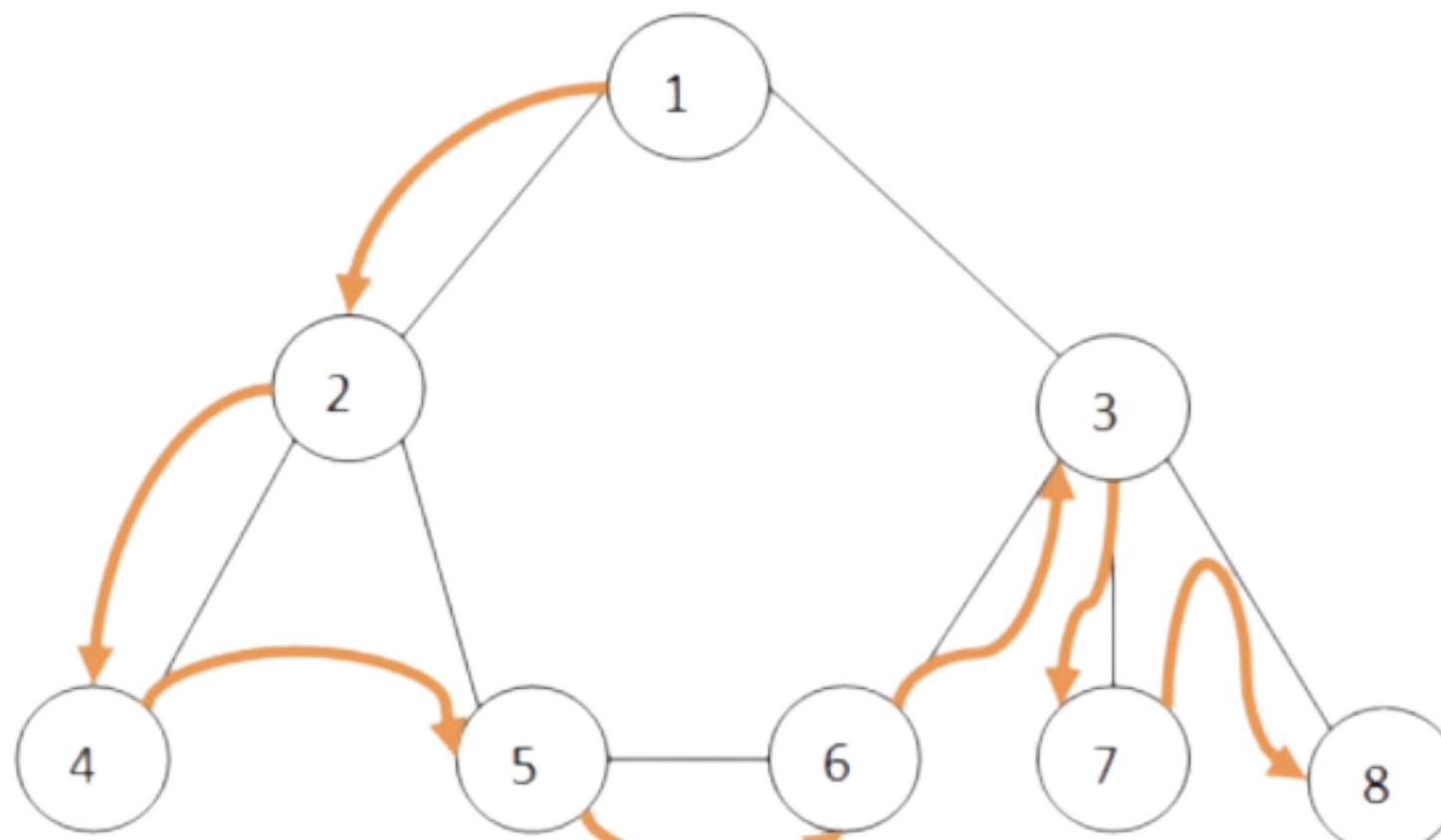
UNIFORM-COST SEARCH

- Expand **least-cost** unexpanded node.
- Implementation: **fringe = queue ordered by path cost.**
- Equivalent to breadth-first if step costs all equal.

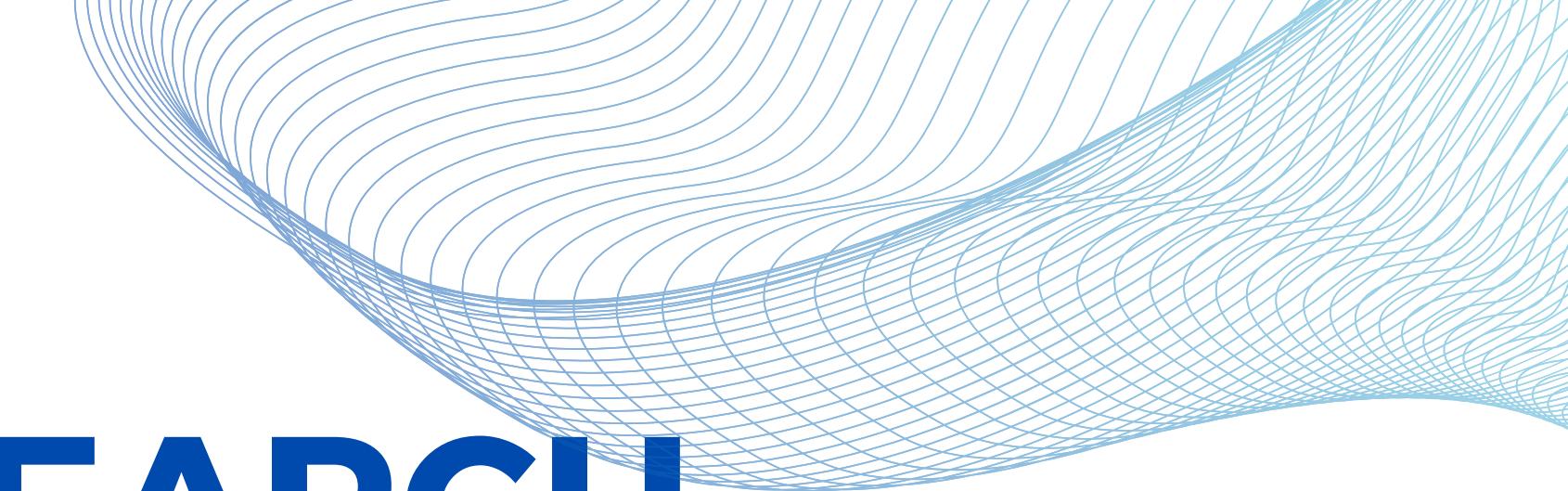
Complete?
Time?
Space?
Optimal?
→HOMEWORK!

DEPTH-FIRST SEARCH

- Expand **deepest** unexpanded node.
- Implementation: **fringe = Last-In-First-Out**, i.e, put successors at front



Complete?
Time?
Space?
Optimal?
→HOMEWORK!



DEPTH-LIMITED SEARCH

- A depth-first search with depth limit

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred?  $\leftarrow$  false
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      result  $\leftarrow$  RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred?  $\leftarrow$  true
      else if result  $\neq$  failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

ITERATIVE DEEPENING SEARCH

- Sometimes used in deep-first search, that finds the best depth limit.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result
```

AVOID REPEATED STATES

- Failure to detect repeated states can turn a linear problem into an exponential one!

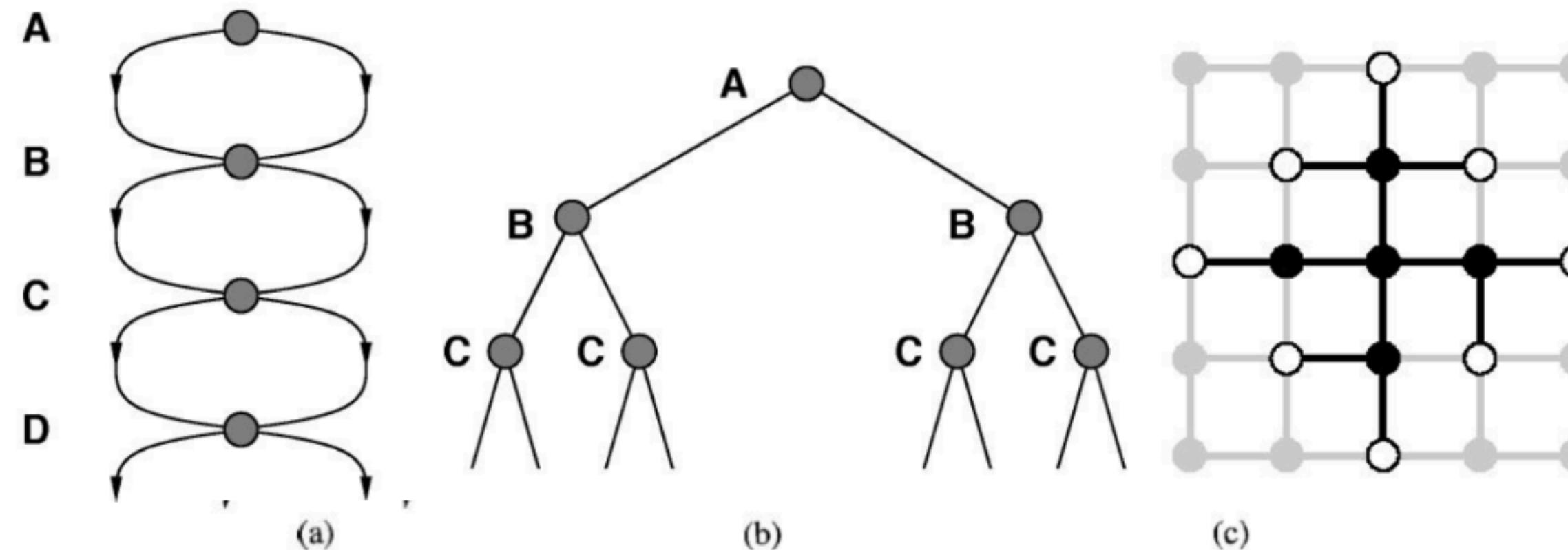


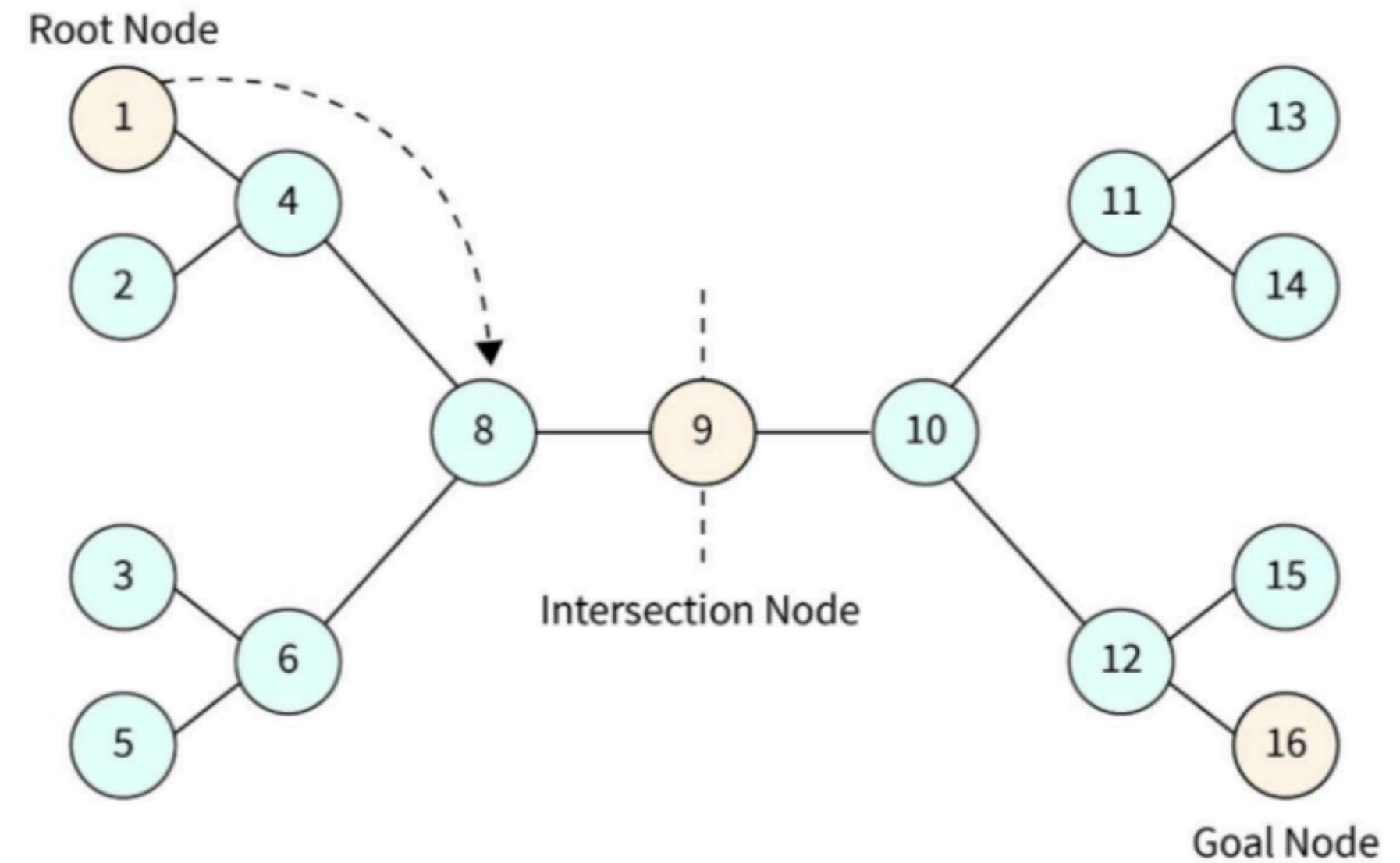
Figure 3.18 State spaces that generate an exponentially larger search tree. (a) A state space in which there are two possible actions leading from A to B, two from B to C, and so on. The state space contains $d + 1$ states, where d is the maximum depth. (b) The corresponding search tree, which has 2^d branches corresponding to the 2^d paths through the space. (c) A rectangular grid space. States within 2 steps of the initial state (A) are shown in gray.

AVOID REPEATED STATES

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

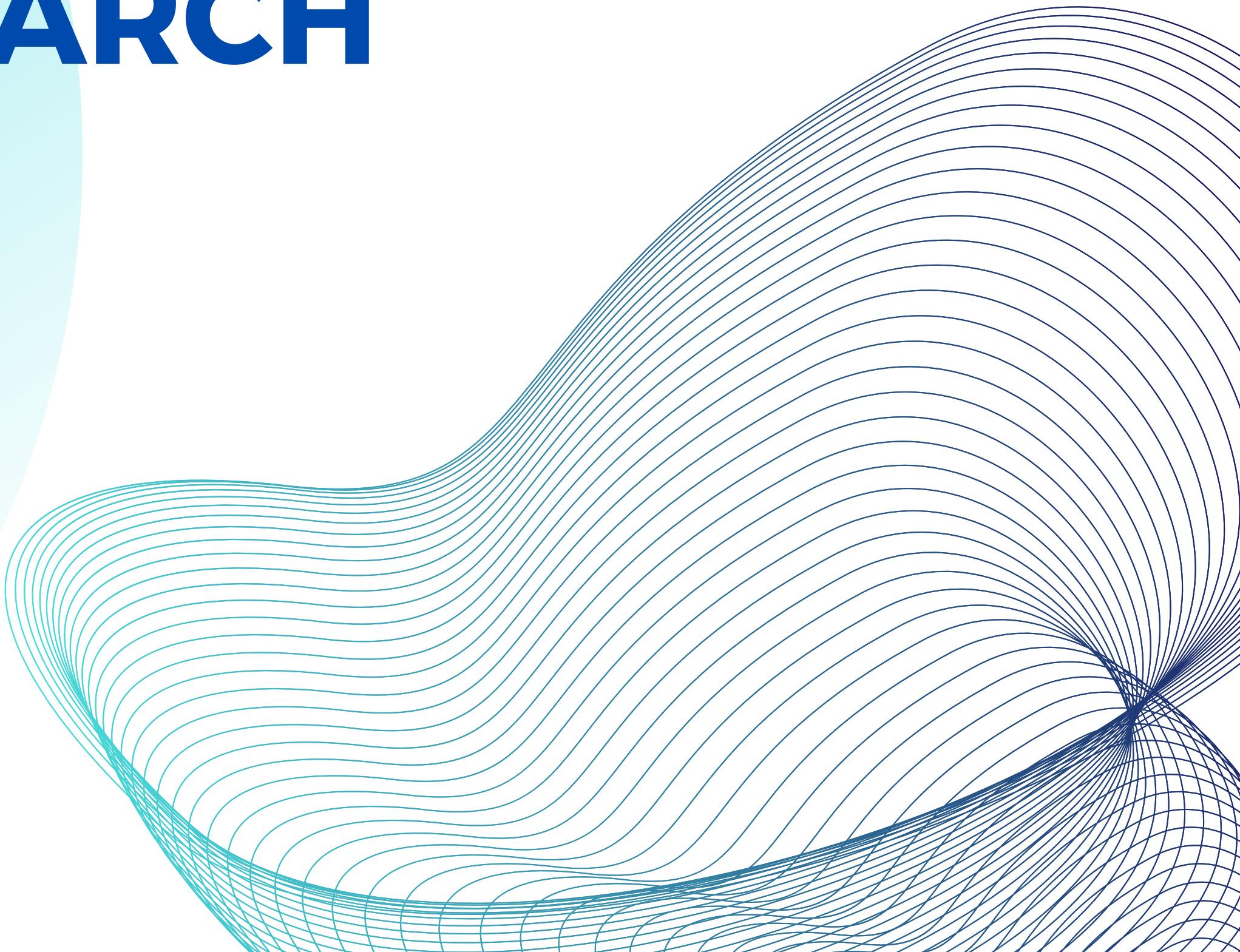
BIDIRECTIONAL SEARCH

- Simultaneously search both forward (from the initial state) and backward (from the goal state)
- Stop when the two searches meet.



INFORMED SEARCH ALGORITHMS

- Best-first search
 - Greedy best-first search
 - A* search
- Heuristics





TREE SEARCH

- A search strategy can be determined by an order of node expansion:

```
function TREE-SEARCH( problem, fringe ) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem ) returns a set of nodes
  successors  $\leftarrow$  the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s  $\leftarrow$  a new NODE
    PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
    add s to successors
  return successors
```

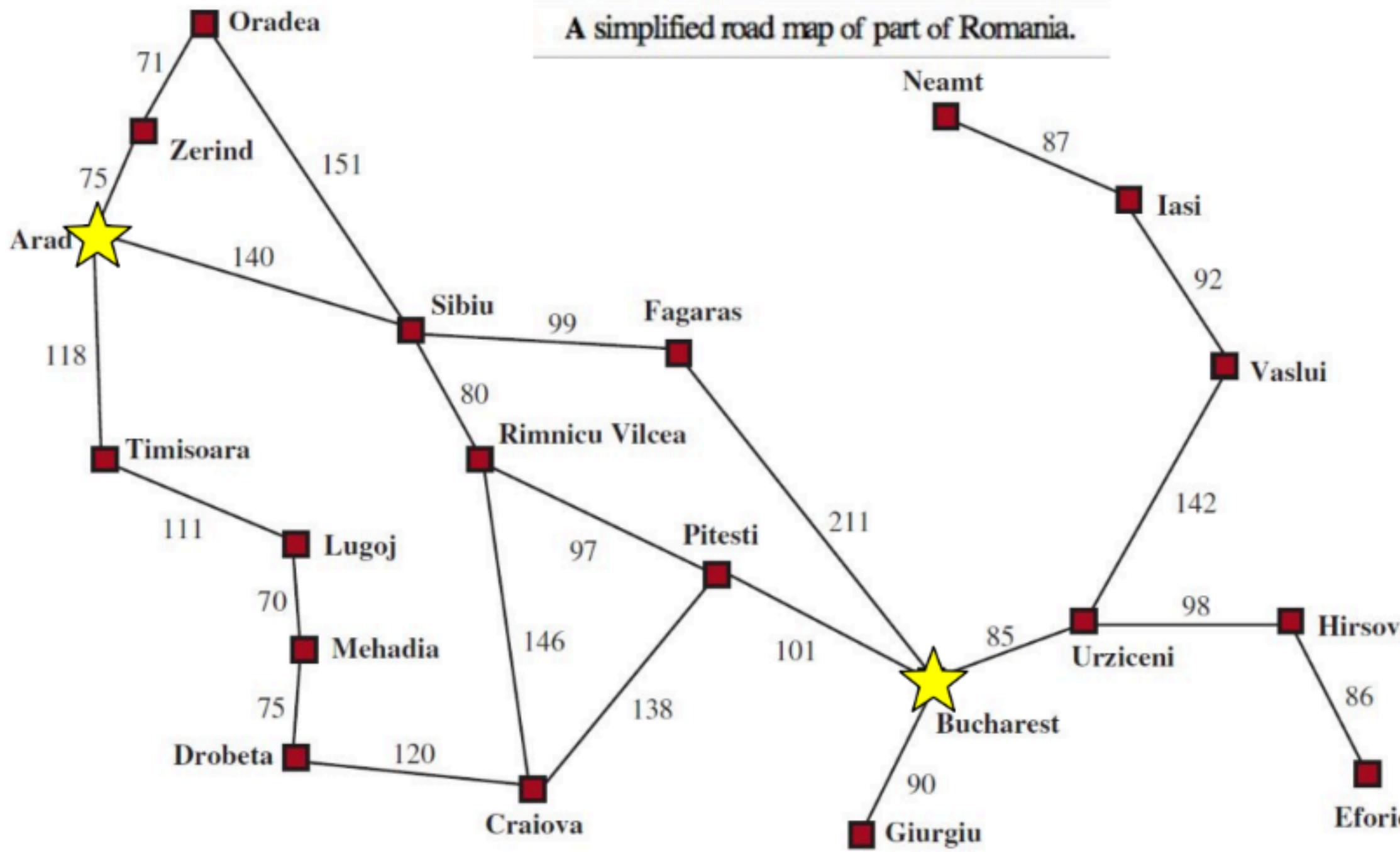
BEST-FIRST SEARCH

- Using an evaluation function $f(n)$ for each node:
 - Estimate of “desirability”
 - Expand the most desirable unexpanded node.
- Order all nodes in fringe in decreasing order of desirability.
- **Special cases:** greedy best-first search, A* search,...



ROMANIAN TRAVELING PROBLEM

- How to go from Arad to Bucharest efficiently?



Straight-line distance to Bucharest (**Value of h_{SLD}**)

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

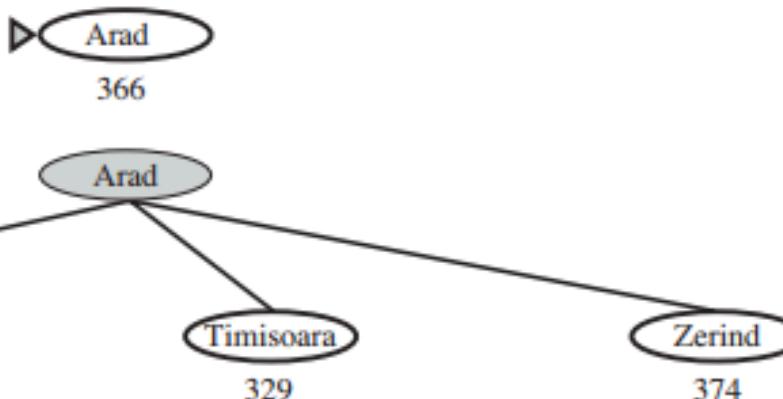
GREEDY BEST-FIRST SEARCH

- Evaluation function:
 $f(n) = h(n)$ (heuristic function)
- It can estimate the cost from step n to our goal.
 - For example: one can choose **$h_{SLD}(n)$** = the **straight-line distance** from the city at step n to Bucharest.
- Greedy best-first search **expands the node that appears to be closest to goal**

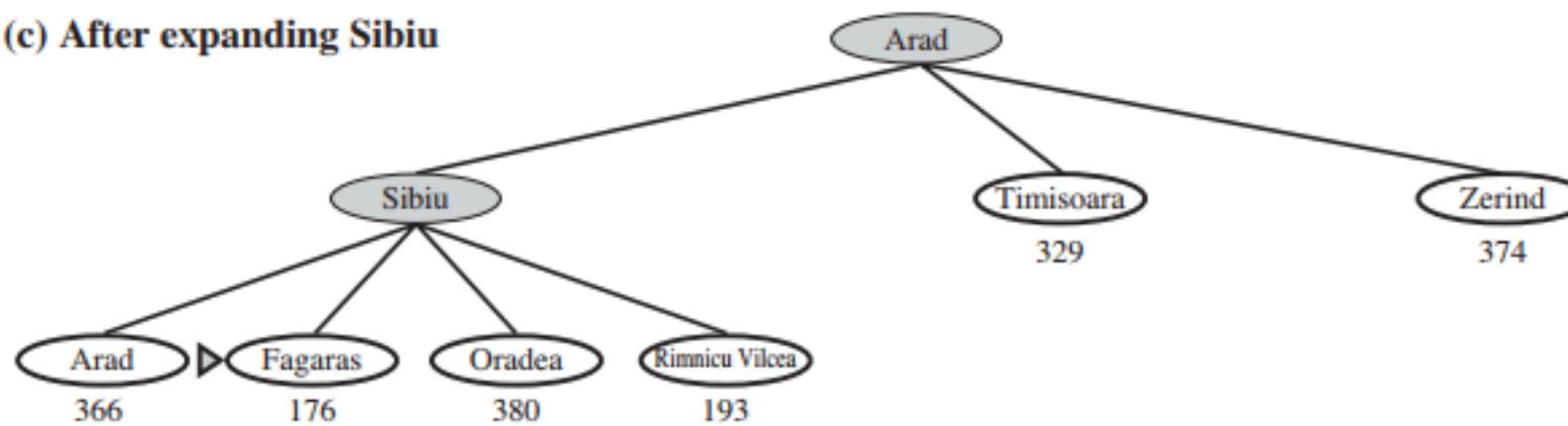
GREEDY BEST-FIRST SEARCH

Example:

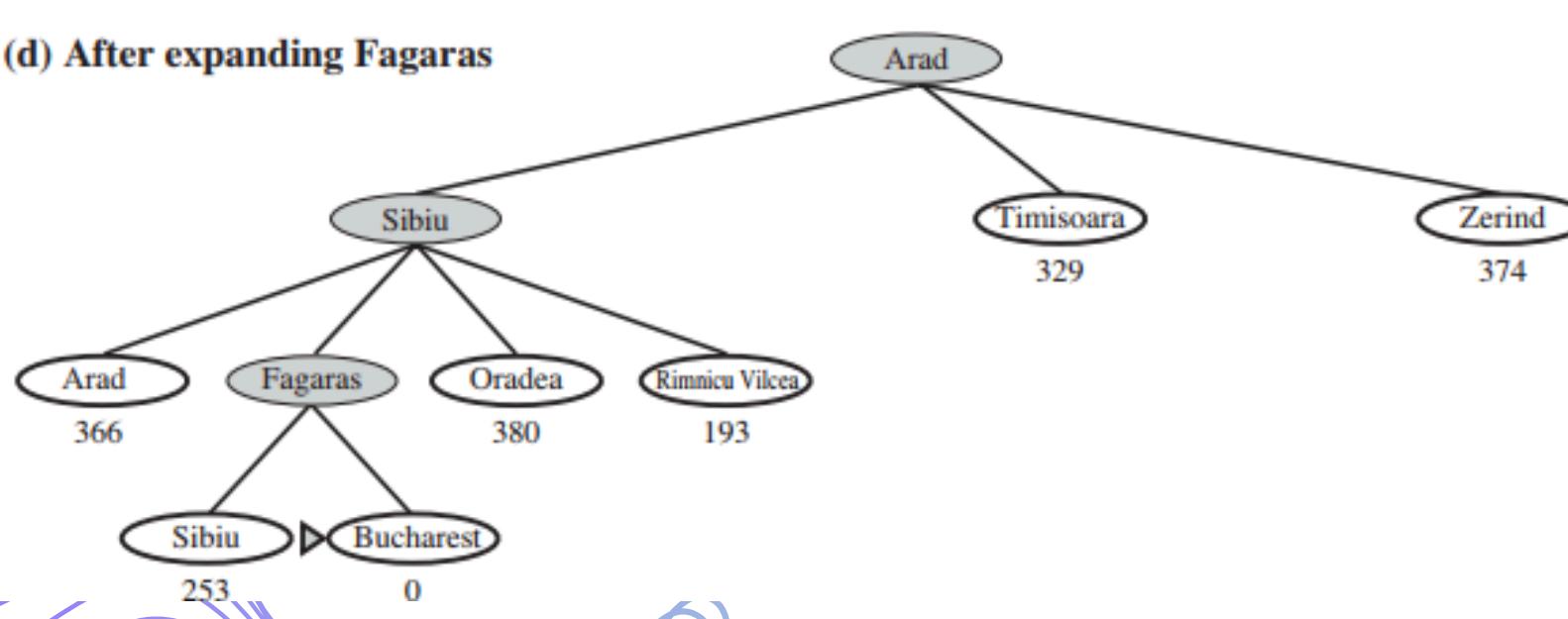
(a) The initial state



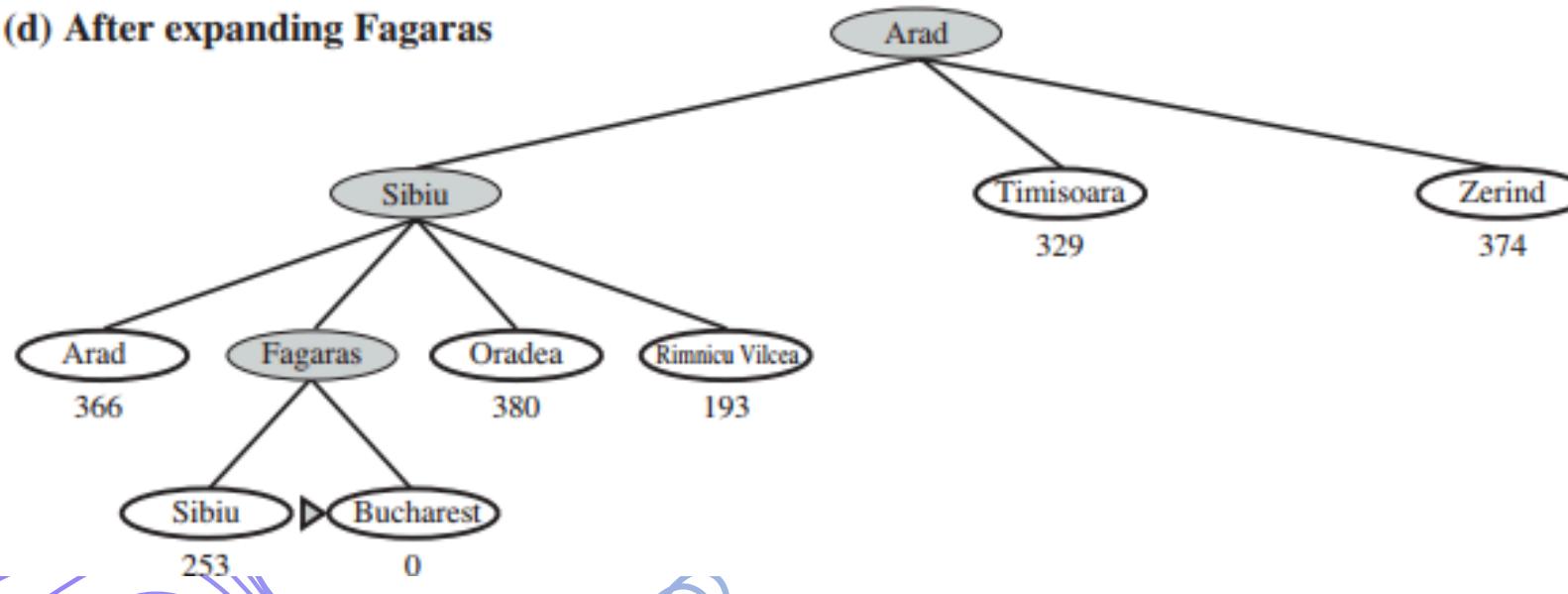
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



GREEDY BEST-FIRST SEARCH

Properties of greedy best-first search:

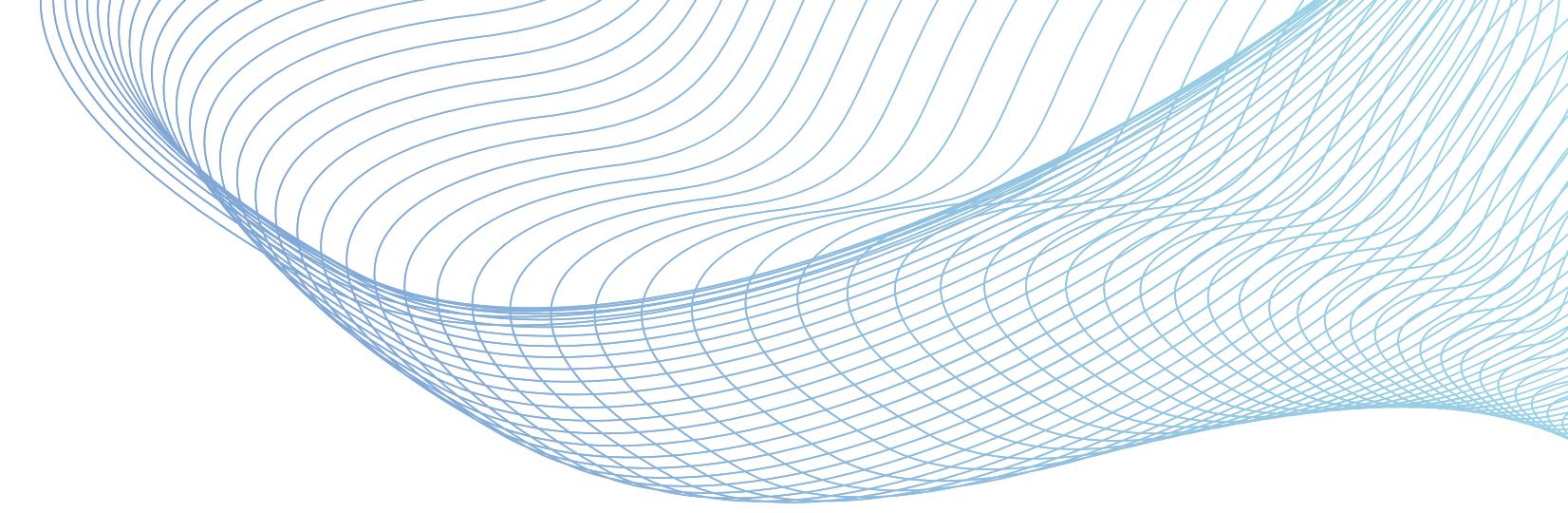
Complete?

Time?

Space?

Optimal?

→HOMEWORK!



A* SEARCH

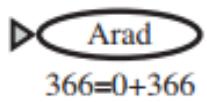
- Avoid expanding paths that are already expensive.
- Minimize the total estimated solution cost.
- Evaluate nodes by: $f(n) = g(n)+h(n)$
 - $g(n)$ = the path cost from the start node to node n.
 - $h(n)$ = the estimated cost of the cheapest path from node n to the goal.
 - $f(n)$ = estimated cost of the cheapest solution through n



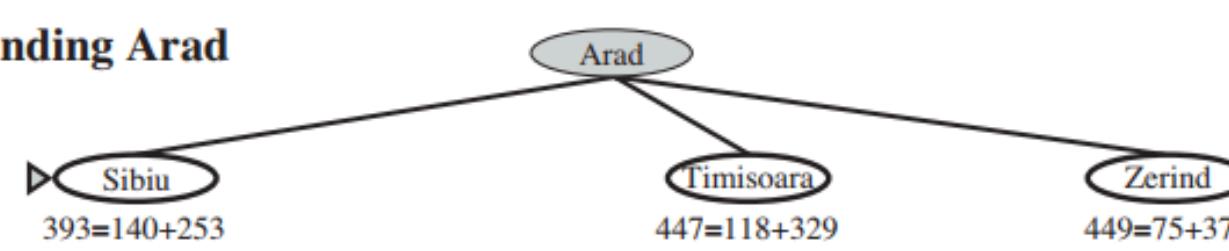
A* SEARCH

Example

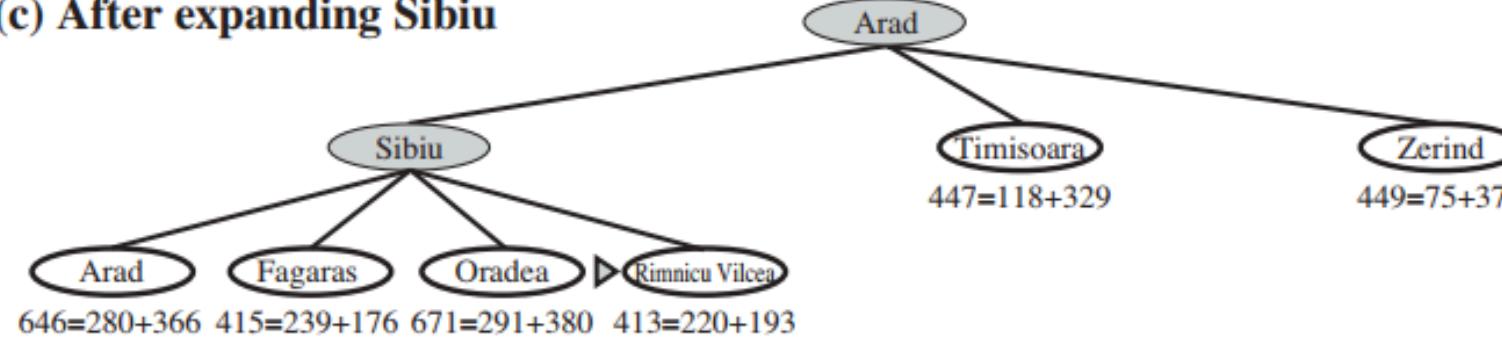
(a) The initial state



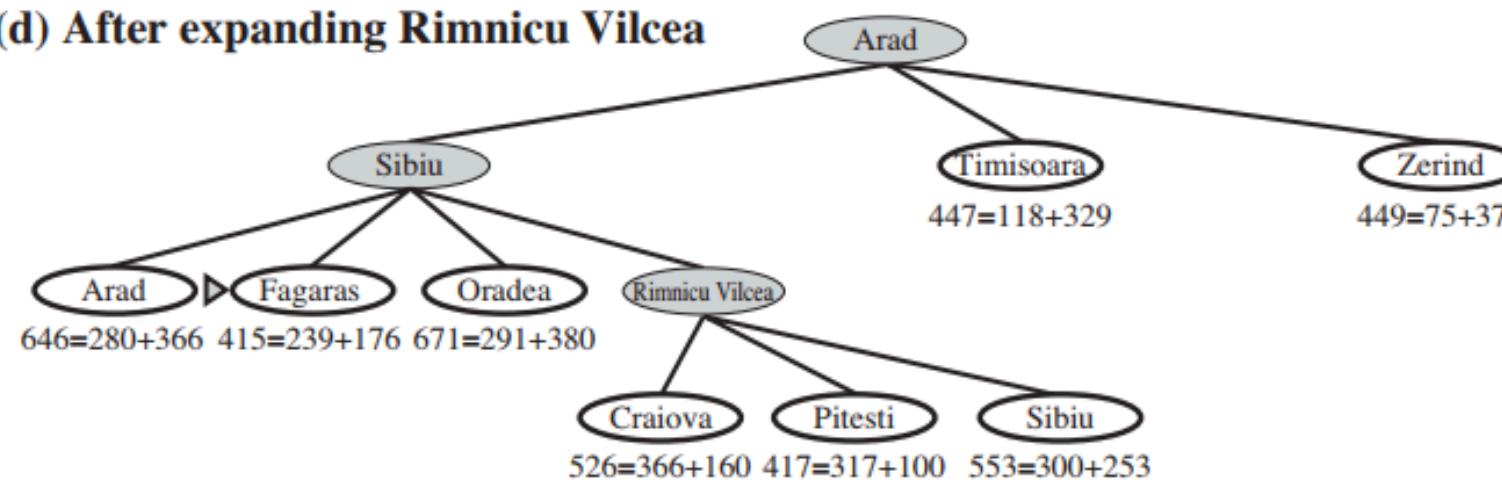
(b) After expanding Arad



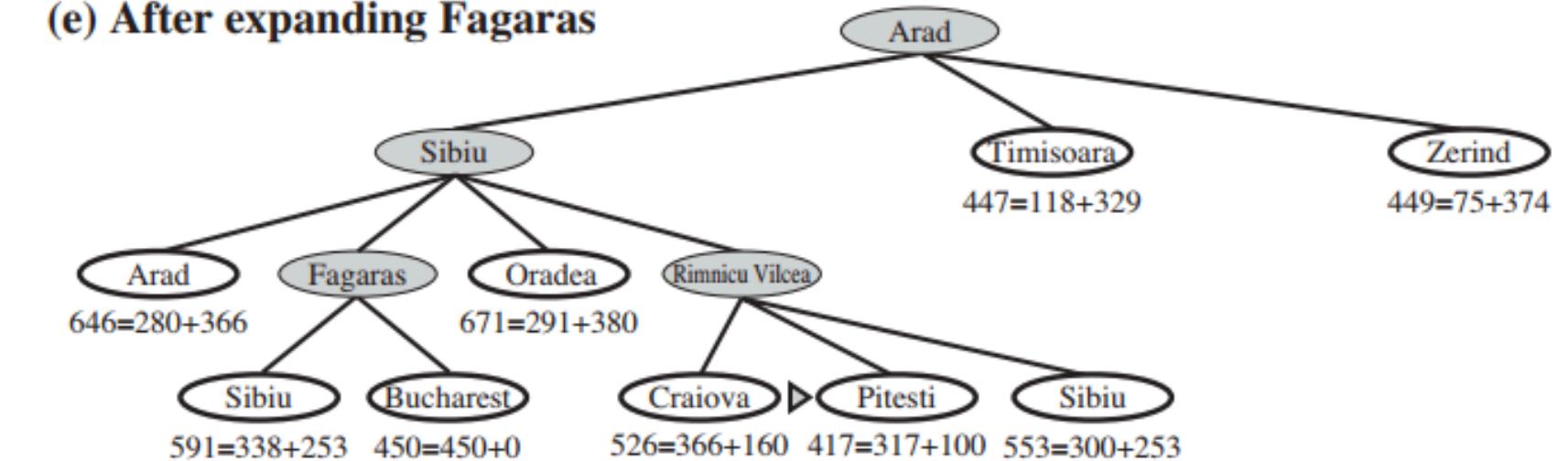
(c) After expanding Sibiu



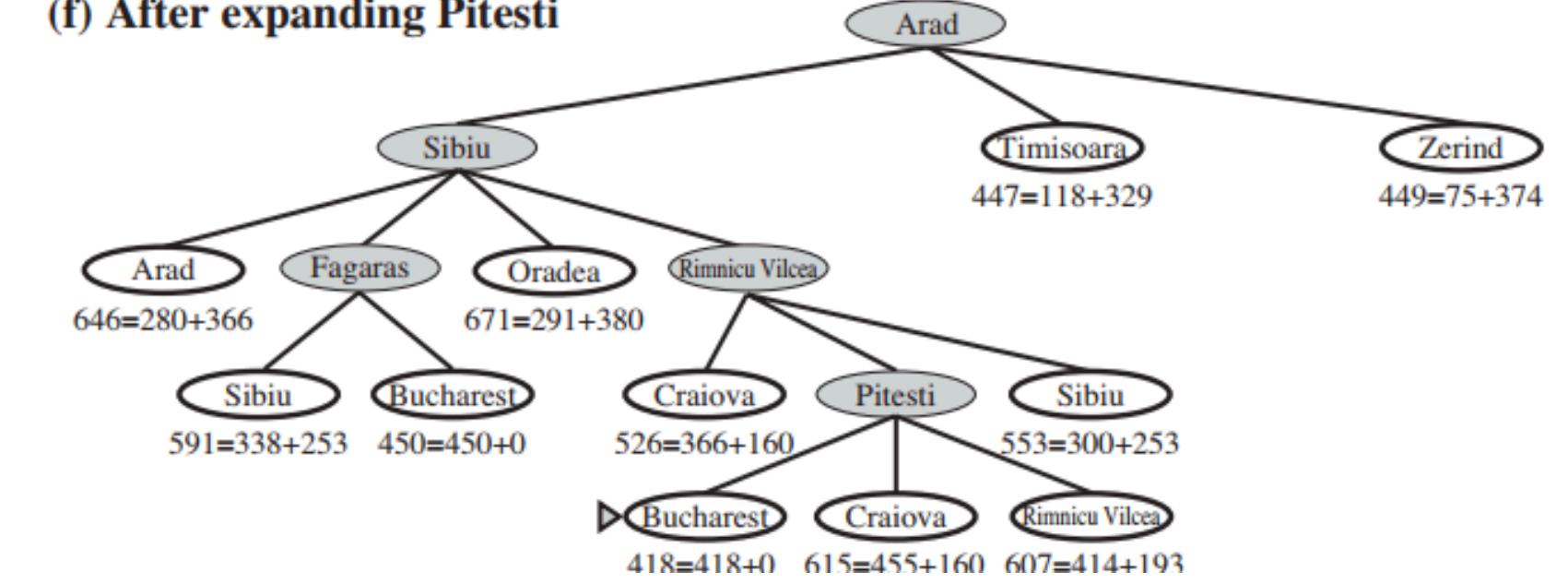
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



ADMISSIBLE HEURISTICS

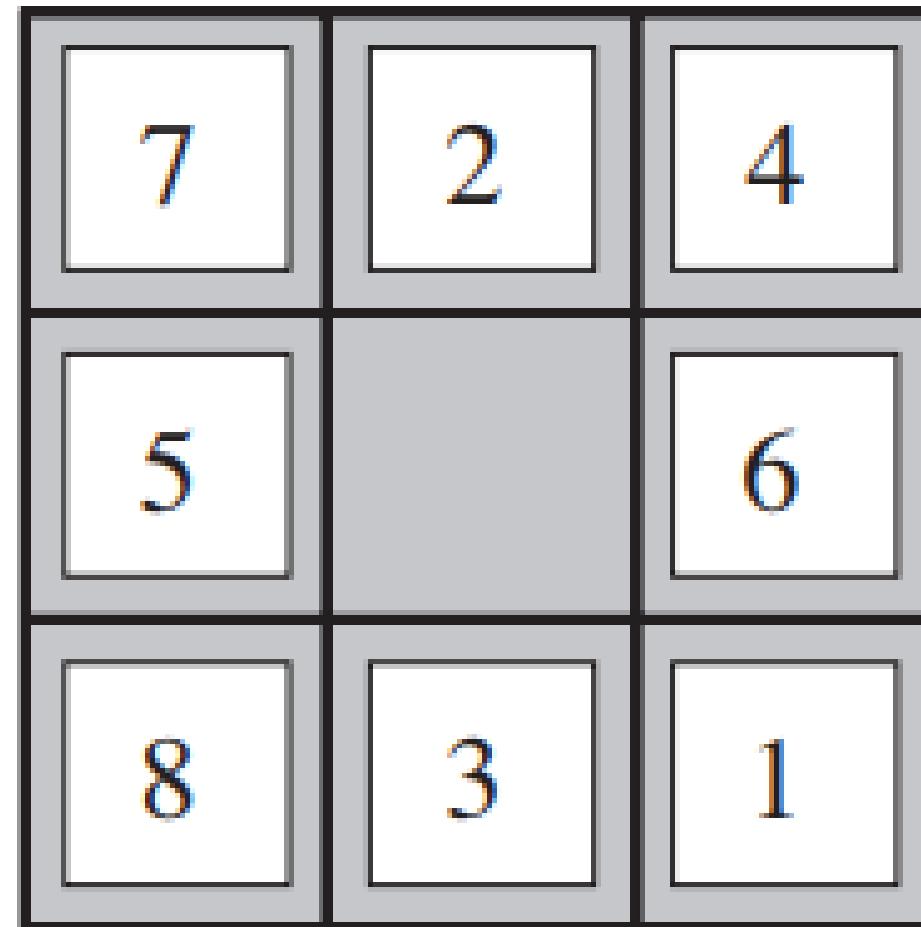
- A heuristic $h(n)$ is admissible if for every node n , $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n .
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance).
- Since $g(n)$ is the exact cost to reach n , $f(n)$ never overestimates the true cost of a solution through n .

ADMISSIBLE HEURISTICS

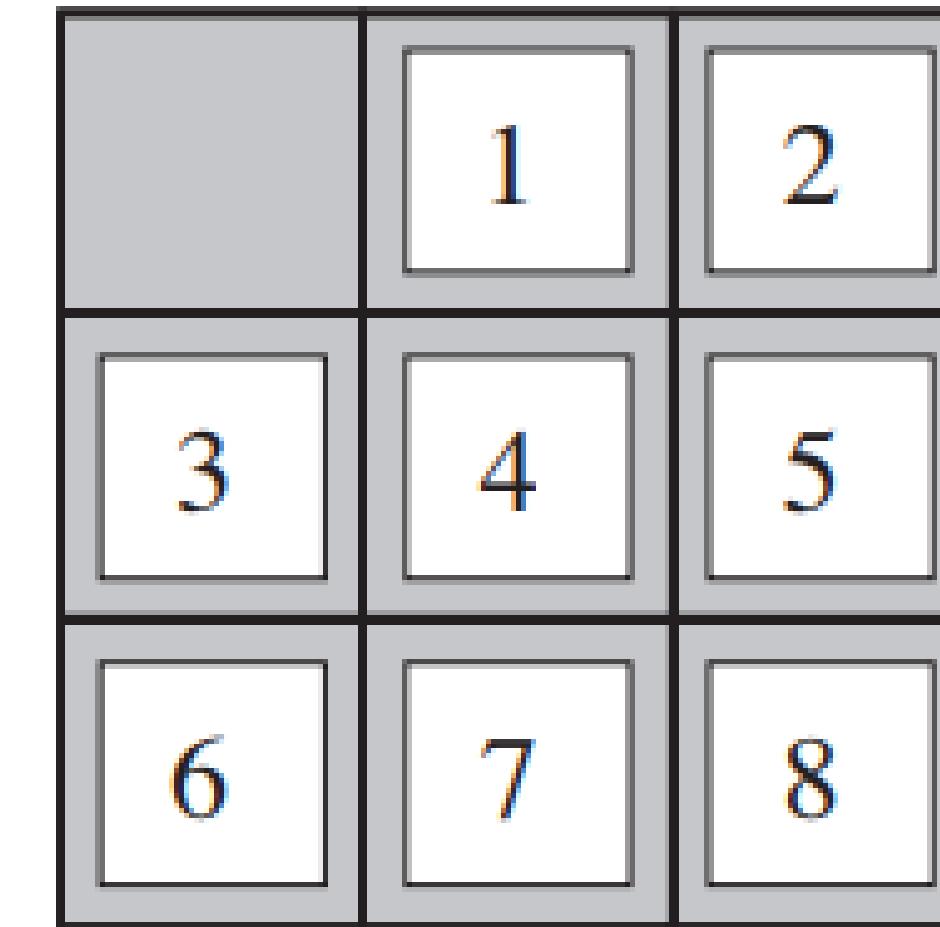
- Theorem: if $h(n)$ is admissible, A* using Tree-Search is optimal.
- PROOF: (HOMEWORK 2)
 - Suppose a suboptimal goal node G_2 appears on the fringe and let the cost of the optimal solution be C^* . Prove that $f(G_2) > C^*$.
 - Consider a fringe node n that is on an optimal solution path. Prove that $f(n) < C^*$.
 - So, G_2 will not be expanded and A* must return an optimal solution.
- If we use the Graph-Search instead of Tree-Search, the proof may fail.
- HOMEWORK 3: EXERCISE 4.4 (BOOK)

ADMISSIBLE HEURISTICS

- Examples: The 8-puzzle



Start State



Goal State

ADMISSIBLE HEURISTICS

- $h_1(n)$ = number of misplaced tiles.
- $h_2(n)$ = total Manhattan distance (i.e., the number of squares from desired location of each tile).
- $h_1(S) = ?$
- $h_2(S) = ?$.

CONSISTENT HEURISTICS

- A heuristic is consistent if for every node n , every successor n' of n generated by an action “ a ”, $h(n) \leq c(n,a,n') + h(n')$
- If h is consistent, we have:
$$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \geq g(n) + h(n) = f(n).$$
 - i.e., $f(n)$ is non-decreasing along any path.
- Theorem: If $h(n)$ is consistent, A* using GraphSearch is optimal.
 - PROOF: (HOMEWORK 4)

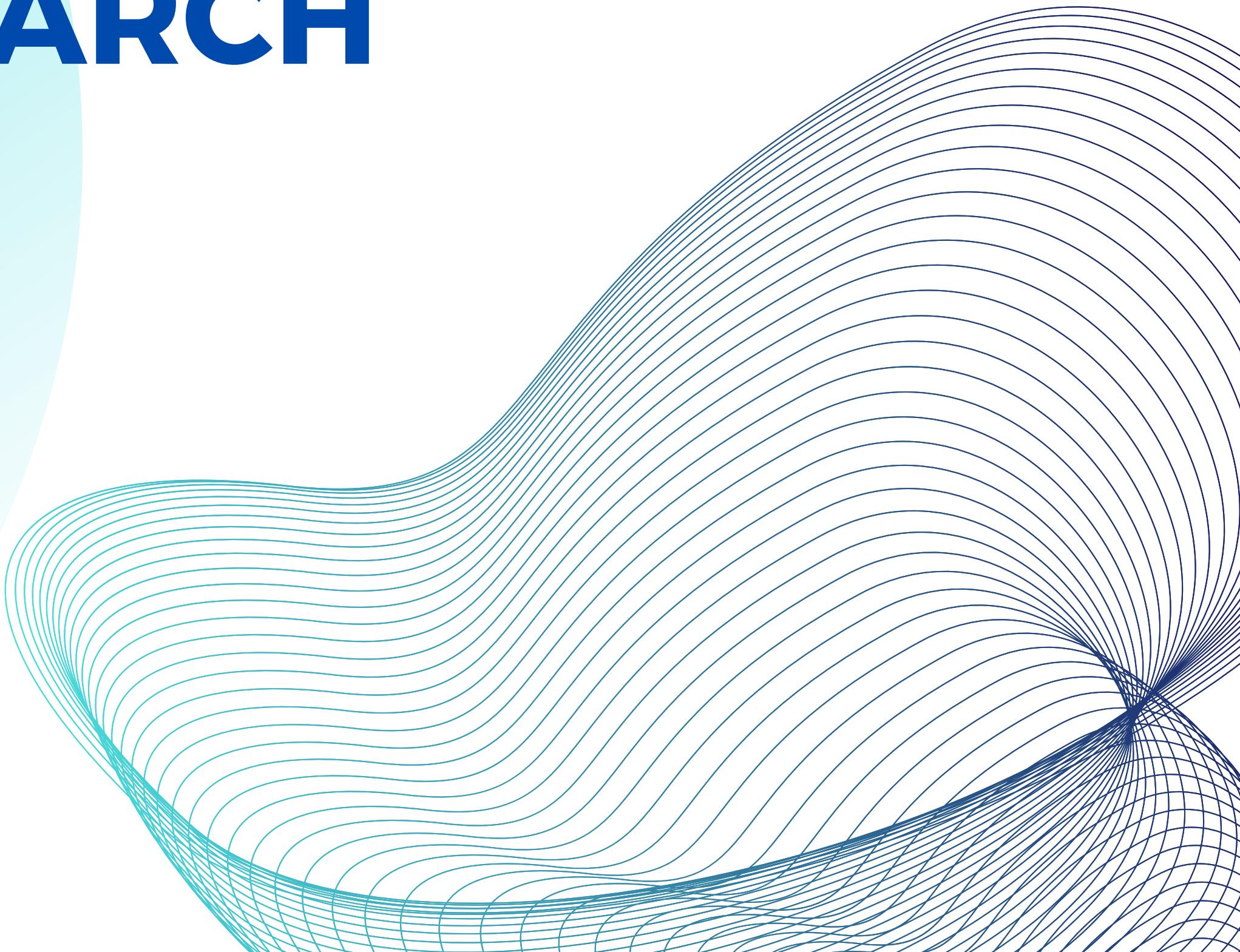
A* SEARCH

Properties of A* search:

Complete?
Time?
Space?
Optimal?
→HOMEWORK 5

INFORMED SEARCH ALGORITHMS

- Relaxed problems
- Local search algorithms



RELAXED PROBLEMS

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, the heuristic function $h_1(n)$ gives the shortest solution

RELAXED PROBLEMS

- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution.
- The original problem can be decomposed into many independent subproblems by the relaxed rules.

LEARNING HEURISTICS

- A heuristic function $h(n)$ is supposed to estimate the cost of a solution beginning from state at node n .
- An agent can construct such a function by devising relaxed problems for which an optimal solution can be found easily.
- Another option is to learn from experience.

LEARNING HEURISTICS

- For example, we solve a lot of 8-puzzles and get an optimal solution for each problem.
- Each such solution can provide examples for which one can learn an heuristic function $h(n)$.
- From these examples, one can construct a function $h(n)$ by an inductive learning that can predict solution costs from other states that arise during search (using neural networks, decision trees,...)

LEARNING HEURISTICS

- Inductive learning methods work best when supplied with features of state that are relevant to its evaluation.
- For example: the feature “numbers of misplaced tiles” → $x_1(n)$. Take 100 randomly generated 8-puzzle configurations and gather statistics on their actual solution costs.

LEARNING HEURISTICS

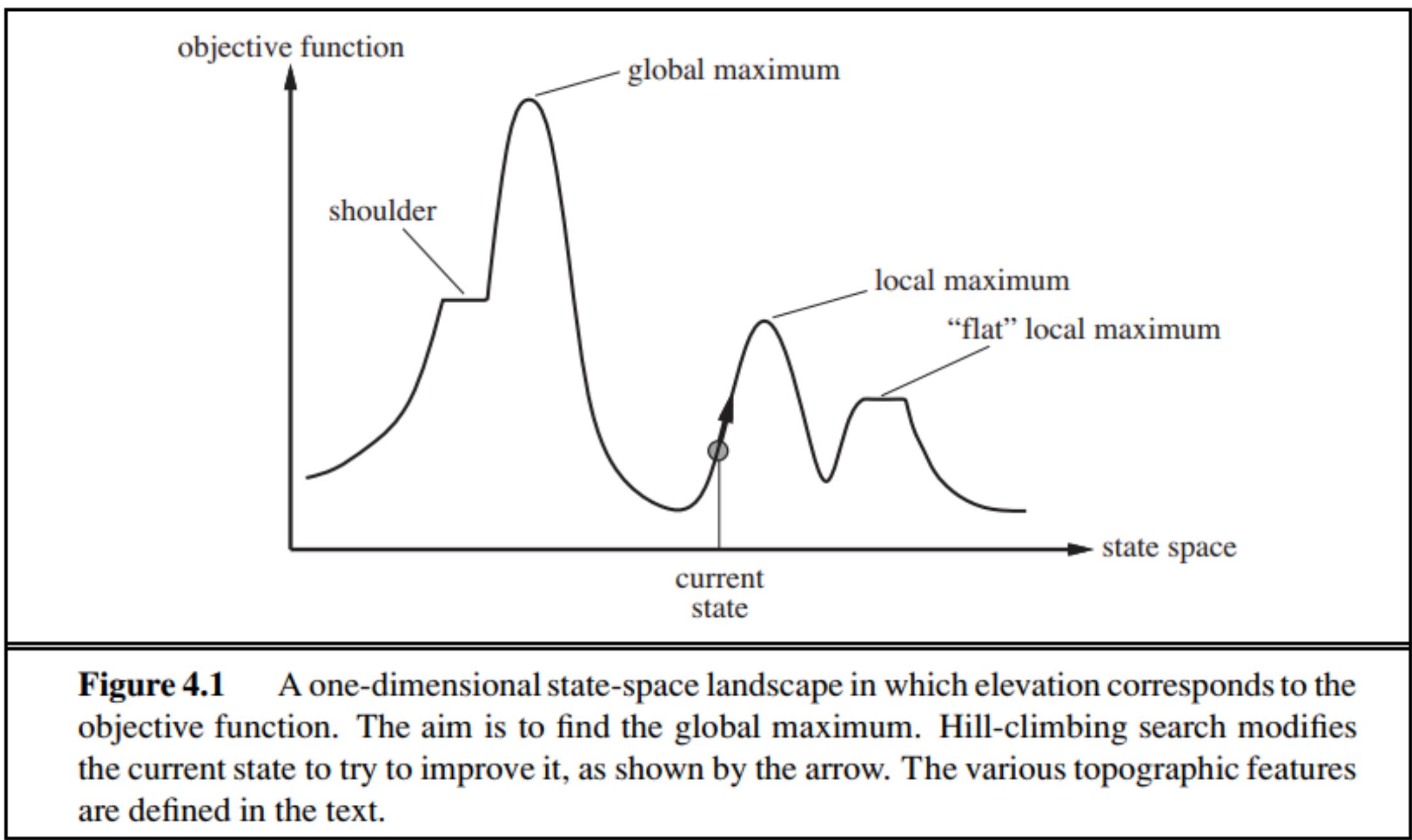
- One may find that when $x1(n)$ is 5, the average solution is around 14,...
- A second feature $x2(n)$: “the number of pairs of adjacent tiles that are adjacent in the goal state as well”.
- Combine the statistical results from $x1(n)$ and $x2(n)$ to predict $h(n)$:
$$h(n) = c1 \cdot x1(n) + c2 \cdot x2(n)$$

LOCAL SEARCH ALGORITHMS

- Use a single current state and generally move only to neighbors of that state.
- Not systematic.
- Two key advantages:
 - use very little memory- usually a constant amount.
 - often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.

LOCAL SEARCH ALGORITHMS

- Use to solve pure optimization problems



HILL-CLIMBING ALGORITHM

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor  $\leftarrow$  a highest-valued successor of current
        if neighbor.VALUE  $\leq$  current.VALUE then return current.STATE
        current  $\leftarrow$  neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h .

SIMULATED ANNEALING SEARCH

- A hill-climbing algorithm can get stuck on a local maximum. Incomplete and fast.
- A purely random walk (moving to a successor chosen uniformly at random from the set of successors. Complete but extremely inefficient.
- Combine hill-climbing algorithm with a random walk in some ways that make efficiency and completeness. → simulated annealing.

SIMULATED ANNEALING SEARCH

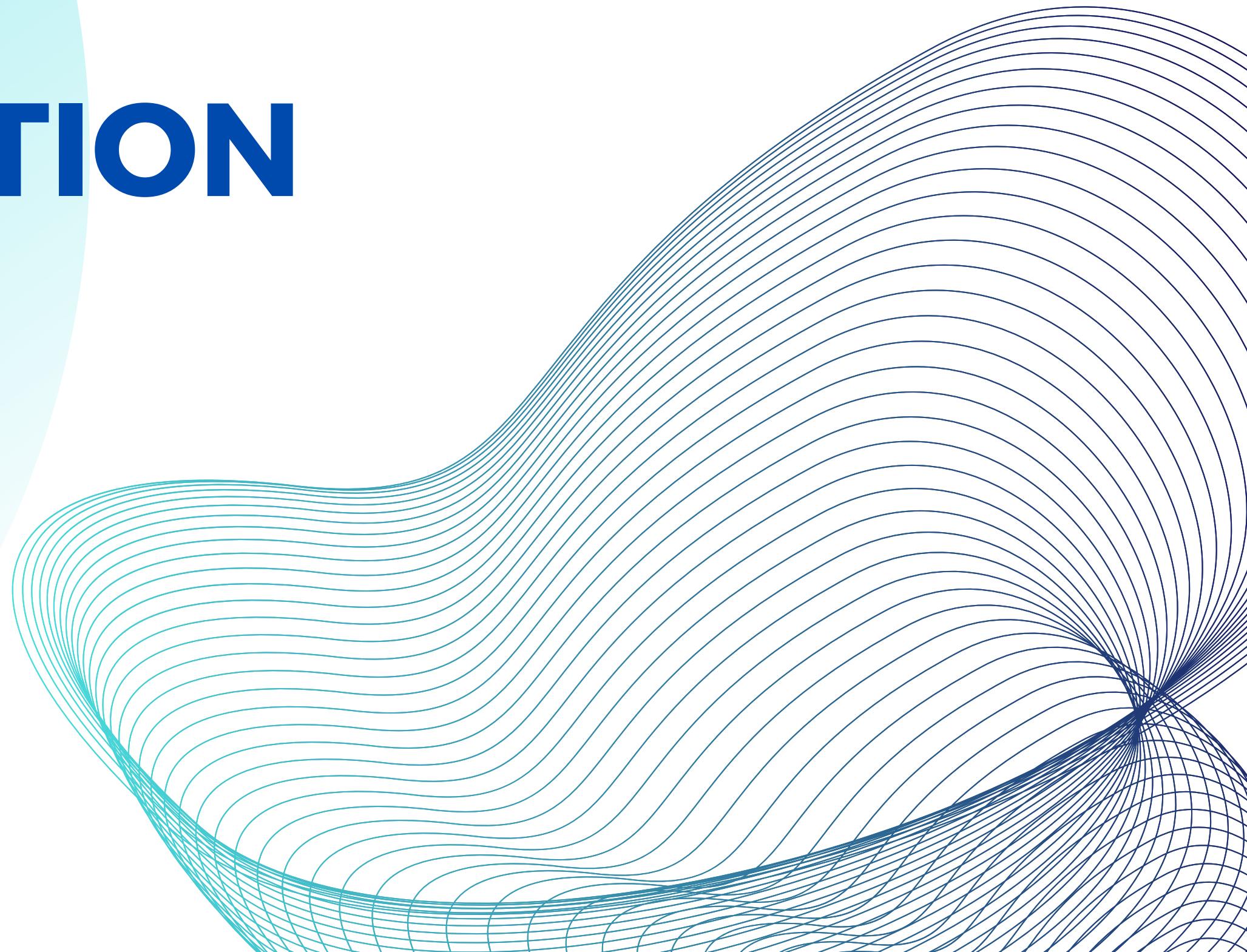
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to “temperature”

    current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
    for t = 1 to  $\infty$  do
        T  $\leftarrow$  schedule(t)
        if T = 0 then return current
        next  $\leftarrow$  a randomly selected successor of current
         $\Delta E \leftarrow$  next.VALUE – current.VALUE
        if  $\Delta E > 0$  then current  $\leftarrow$  next
        else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature *T* as a function of time.

KNOWLEDGE REPRESENTATION

- Introduction
- Knowledge-based agent
- Kinds of knowledge
- Knowledge representation



INTRODUCTION

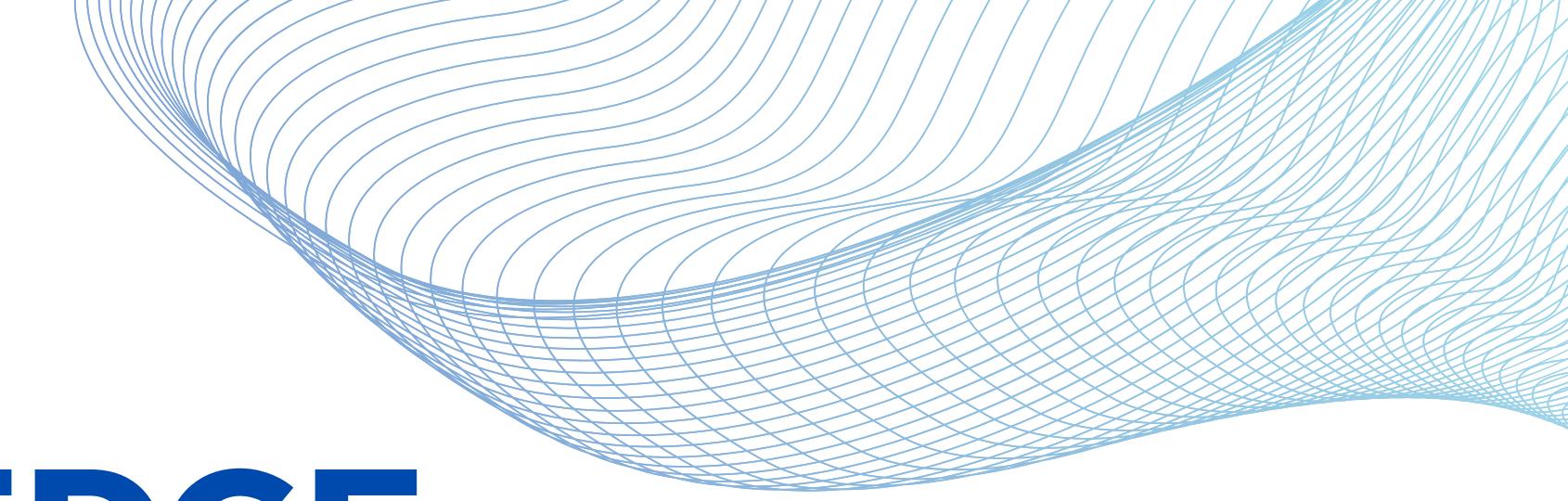
- Con người có thể cảm nhận thế giới quan bằng các giác quan, sử dụng các tri thức tích luỹ được và bằng các lập luận, suy diễn để đưa ra những hành động thích hợp.
- Các hệ thông minh (intelligent agent) cần phải có tri thức về thế giới hiện thực và môi trường xung quanh để đưa ra những quyết định đúng đắn

KNOWLEDGE-BASED AGENTS

- Knowledge-based agent: là hệ tri thức chứa các cơ sở tri thức (knowledge base).
- Cơ sở tri thức là tập hợp các tri thức được biểu diễn dưới dạng nào đó.
- Mỗi khi nhận được thông tin đưa vào (input data), các agent phải có khả năng suy diễn để đưa ra những phương án chính xác, hợp lý.

KNOWLEDGE-BASED AGENTS

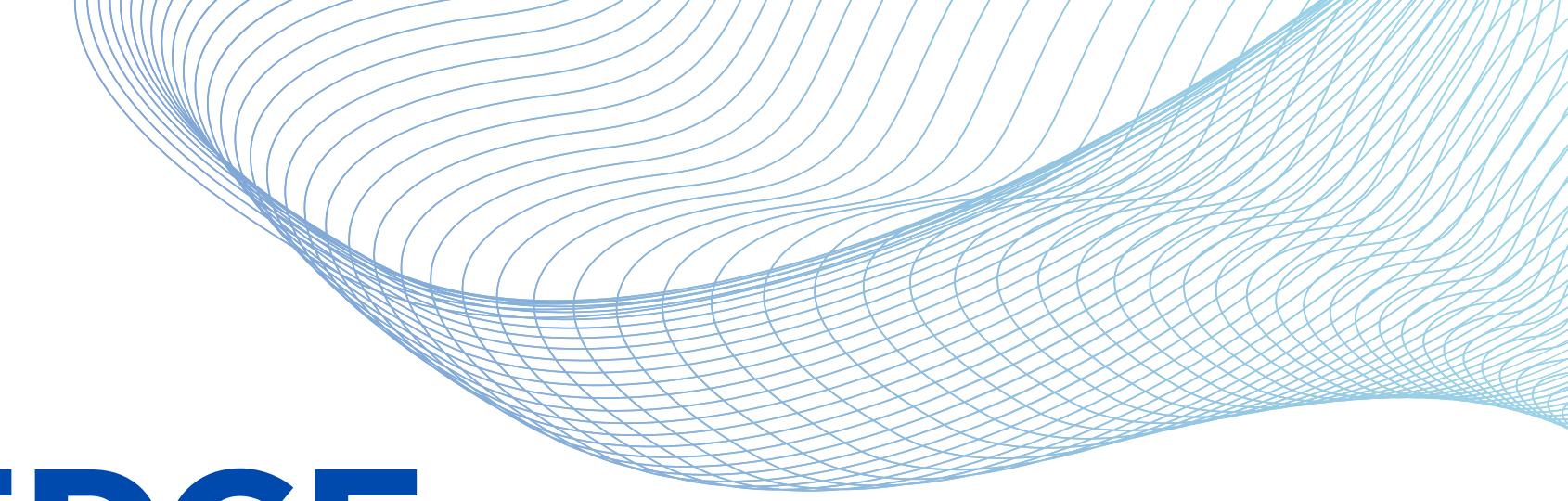
- Hệ tri thức cần được trang bị một cơ chế suy diễn.
- Đối với hệ thống thông minh giải quyết một vấn đề nào thì cơ sở tri thức sẽ chứa các tri thức tương ứng.
- Tri thức (knowledge) là khái niệm trừu tượng.
- Theo từ điển Oxford, “*knowledge is information and skills acquired through experience or education*”.



KINDS OF KNOWLEDGE

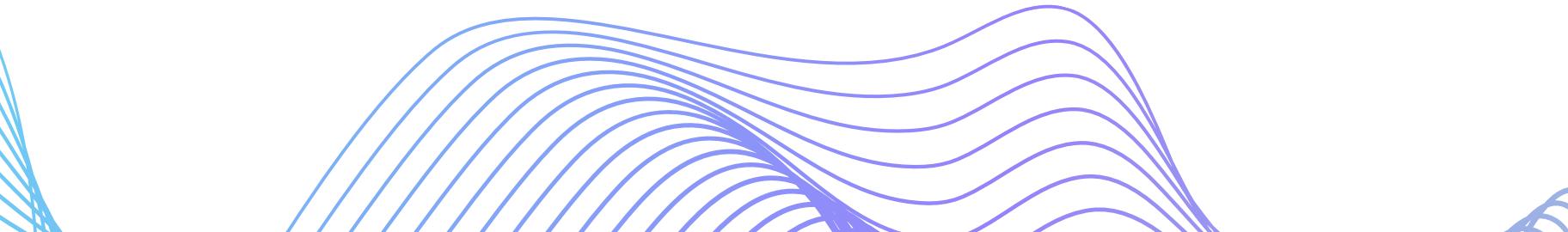
- **Tri thức lý thuyết** (*theoretical or a priori knowledge*): là tri thức đạt được mà không cần quan sát thế giới quan.
- **Tri thức thực tiễn** (*empirical knowledge*): tri thức đạt được bằng những quan sát và tương tác với môi trường xung quanh
- **Tri thức mô tả** (*declarative knowledge*): bao gồm những từ ngữ mô tả chính xác về một sự vật, hiện tượng

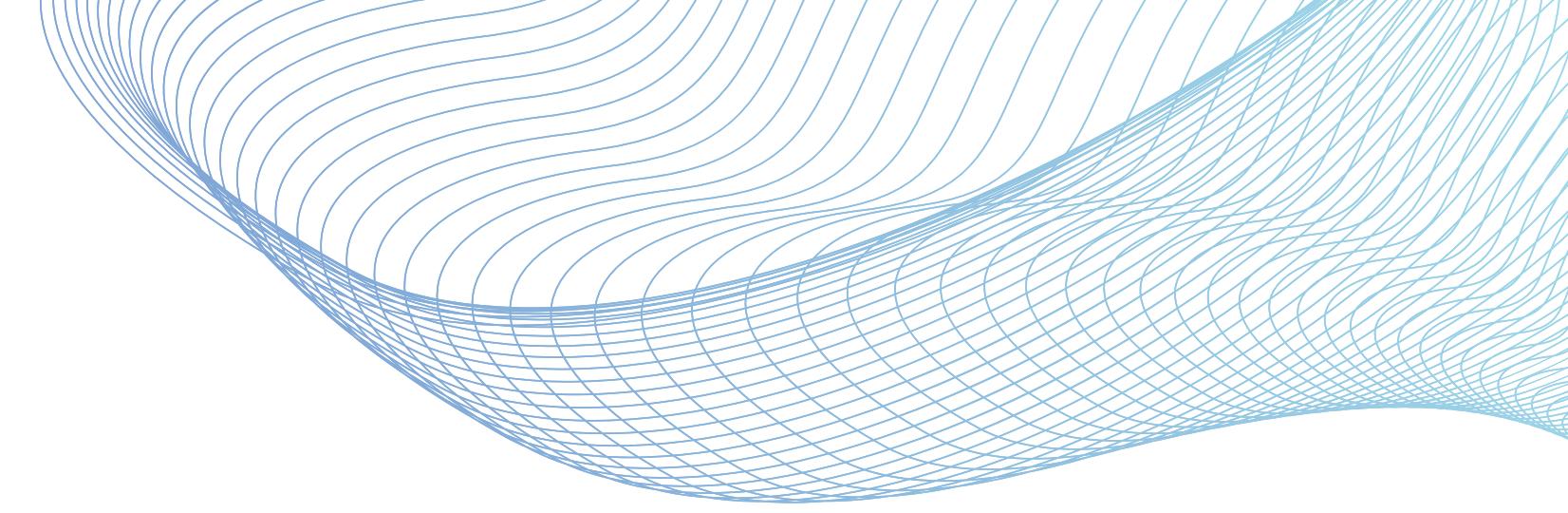




KINDS OF KNOWLEDGE

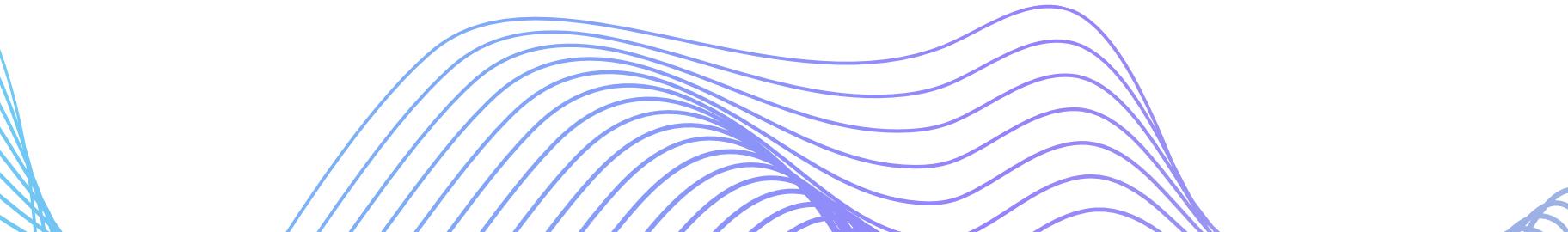
- **Tri thức quy trình** (*procedural knowledge*): bao gồm những từ ngữ dùng để mô tả một quá trình (process) nào đó.
- **Tri thức heuristic** (*heuristic knowledge*): là tri thức nông cạn do không đảm bảo chính xác hoặc tối ưu khi giải quyết vấn đề. Thường được coi như mèo nhầm dẫn dắt quá trình lập luận.

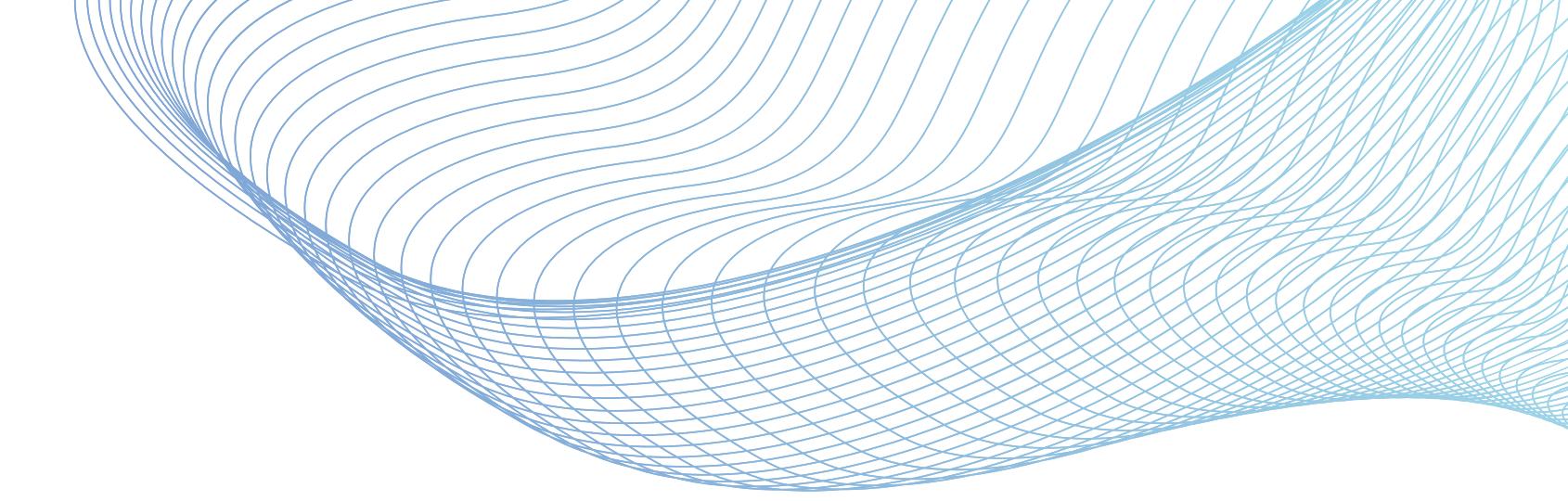




EXPERT SYSTEM

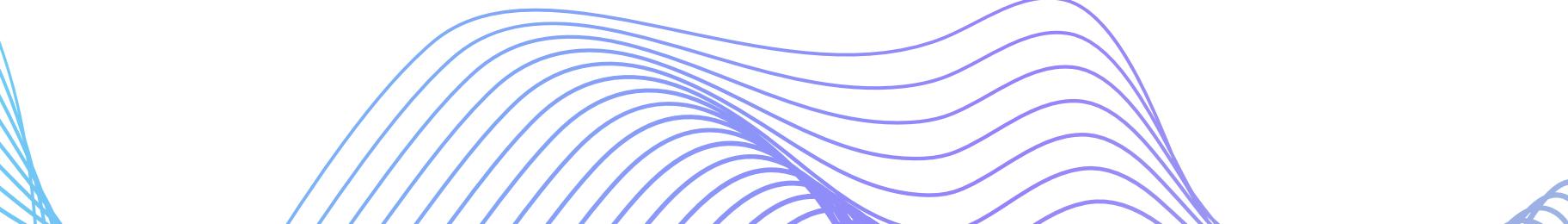
- **Chuyên gia** (expert) là những người có kiến thức sâu sắc về một vấn đề nào đó và có thể giải quyết tốt những việc mà ít ai làm được.
- **Hệ chuyên gia** (expert system) là chương trình máy tính có thể thực hiện những công việc trong một lĩnh vực nào đó như một chuyên gia thực thụ.





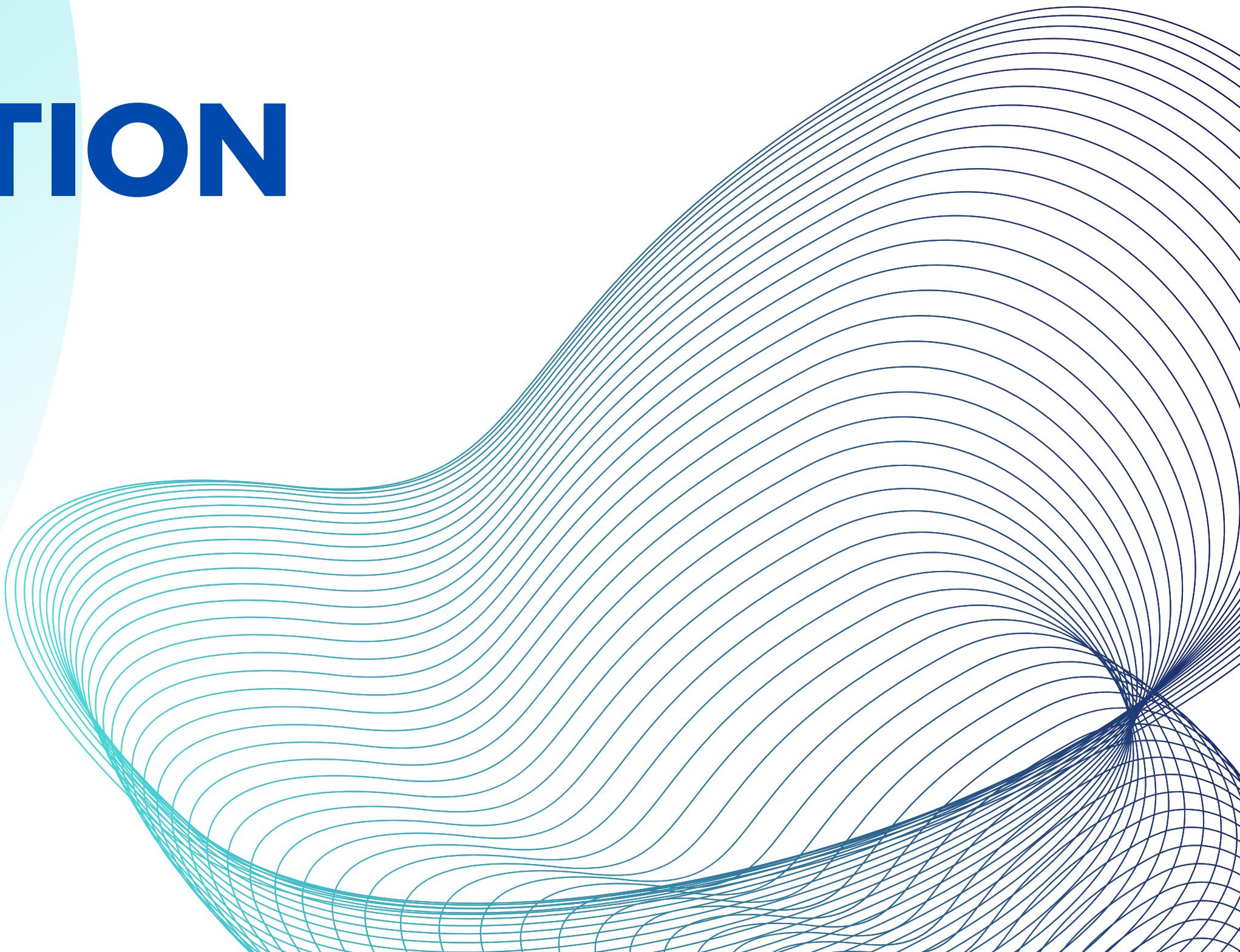
EXPERT SYSTEM

- Là các hệ tri thức dựa trên luật suy diễn phức tạp.
- Áp dụng trên rất nhiều lĩnh vực trong công nghiệp như marketing, nông nghiệp, y tế, điện lực,...



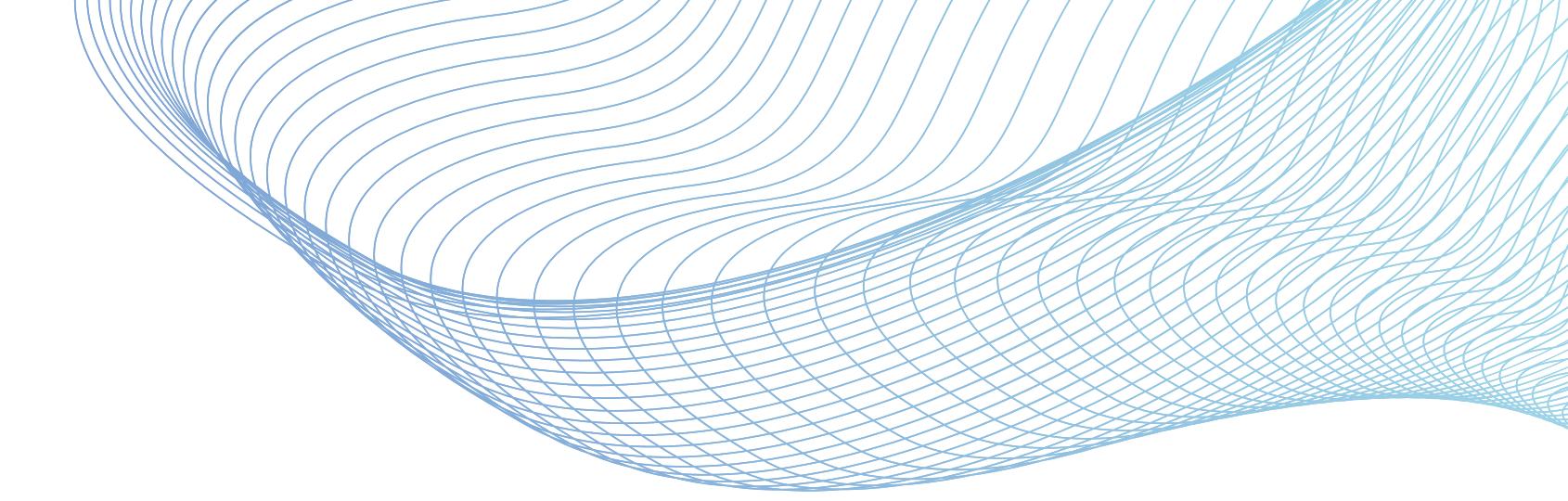
KNOWLEDGE REPRESENTATION

- Logic mệnh đề (propositional logic)
- Logic vị từ cấp một (first-order predicate logic).



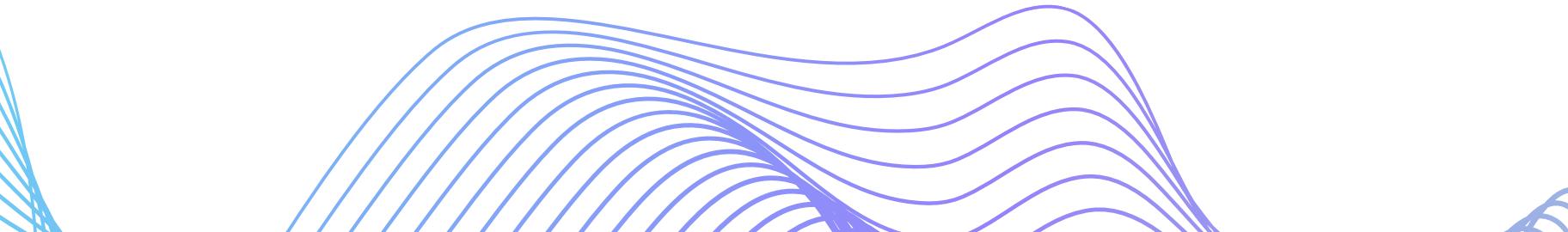
PROPOSITIONAL LOGIC

- **Logic mệnh đề** là công cụ logic trong đó các mệnh đề được mã hoá cho một biến hoặc hằng, còn các biểu thức là sự liên kết có nghĩa giữa các biến và các toán tử logic nhất định.
- Ví dụ: “Nếu tôi cố gắng làm bài tập (A) thì tôi sẽ thi tốt (B)” được mô tả như sau: $A \rightarrow B$



PROPOSITIONAL LOGIC

- Tri thức sẽ được mô tả dưới dạng các mệnh đề trong ngôn ngữ biểu diễn tri thức.
- Gồm hai thành phần cơ bản: cú pháp và ngữ nghĩa.
- Cú pháp của một ngôn ngữ bao gồm các ký hiệu và quy tắc liên kết các ký hiệu (luật cú pháp) để tạo thành các câu (công thức) trong ngôn ngữ.



PROPOSITIONAL LOGIC

- Ngữ nghĩa của ngôn ngữ cho phép chúng ta xác định ý nghĩa của các câu trong một miền nào đó của thế giới thực.
- Ví dụ: $1+2+3+\dots+n$
- Ngoài cú pháp và ngữ nghĩa, ngôn ngữ biểu diễn tri thức cần được cung cấp cơ chế suy diễn, giúp chúng ta tìm ra một công thức mới từ một tập nào đó các công thức.

PROPOSITIONAL LOGIC

- Ngôn ngữ biểu diễn tri thức = cú pháp + ngữ nghĩa+ cơ chế suy diễn.
- Một ngôn ngữ biểu diễn tri thức tốt cần có khả năng biểu diễn rộng, tức là mô tả được hầu hết điều chúng ta muốn.
- Hiệu quả đi đến kết luận + thủ tục suy diễn đòi hỏi ít thời gian và không gian nhớ.
- Càng gần với ngôn ngữ tự nhiên càng tốt

CÚ PHÁP

- Cú pháp của logic mệnh đề đơn giản và cho phép xây dựng các công thức.
- Gồm tập các ký hiệu và tập các luật xây dựng công thức.
- Các ký hiệu:
 - Hằng logic: true và false
 - Các ký hiệu mệnh đề: P, Q, R,...
 - Các phép kết nối logic: \wedge , \vee , \rightarrow , \leftrightarrow
 - Các dấu mở ngoặc và đóng ngoặc

CÚ PHÁP

- Các quy tắc xây dựng công thức:

- $A \vee B$
- $A \wedge B$
- $A \leftrightarrow B$
- $A \rightarrow B \dots$

PROPOSITIONAL LOGIC

- Ngữ nghĩa của logic mệnh đề giúp ta xác định được ý nghĩa thực sự của các công thức.
- Bất kỳ một sự kết hợp các ký hiệu mệnh đề với các sự kiện trong thế giới thực được gọi là minh họa (interpretation).
- Thường được gán một giá trị chân lý True hoặc False.

PROPOSITIONAL LOGIC

- Bảng chân lý giúp ta xác định ngữ nghĩa câu phức hợp.
- Một công thức được gọi là thoả mãn (satisfiable) nếu nó đúng trong một minh họa nào đó.
- Ví dụ:
 $(P \vee Q) \wedge R$ thoả mãn vì nó có giá trị True khi {P: False, Q: True, R: True}

PROPOSITIONAL LOGIC

- Một công thức gọi là vững chắc (valid) nếu nó đúng trong mọi minh họa.
- Một công thức được gọi là không thoả được nếu nó sai trong mọi minh họa.
- Ta gọi một mô hình (model) của một công thức là một minh họa để công thức đúng trong trường hợp đó

PROPOSITIONAL LOGIC

- Cách xác định một công thức có vững chắc (thoả mãn, không thoả mãn): lập bảng chân trị.
- Một tập các công thức $G = (G_1, \dots, G_n)$ là vững chắc (thoả mãn, không thoả mãn) nếu hội của chúng $G_1 \wedge G_2 \wedge \dots \wedge G_n$ là vững chắc (thoả mãn, không thoả mãn).
- Một mô hình của G là mô hình của công thức $G_1 \wedge G_2 \wedge \dots \wedge G_n$.

PROPOSITIONAL LOGIC

- Hai công thức được gọi là tương đương nếu chúng có cùng giá trị chân lý trong mọi trường hợp. Ta chỉ hai công thức A và B tương đương, ta viết $A \equiv B$.
- Luật De Morgan, giao hoán, kết hợp, phân phối.

PROPOSITIONAL LOGIC

- Để viết chương trình trên máy tính thao tác các công thức, chúng ta thường chuẩn hoá chúng về dạng chuẩn tắc.
- Một công thức được gọi là chuẩn tắc nếu nó là hội của các câu phức tạp (clause). Một câu phức tạp có dạng $A_1 \vee A_2 \vee \dots \vee A_n$, trong đó A_i là các câu đơn (literal).

PROPOSITIONAL LOGIC

Phương pháp chuẩn hoá:

- Bỏ các dấu kéo theo bằng các luật.
- Chuyển các dấu phủ định bằng luật De Morgan.
- Áp dụng luật phân phối, thay công thức $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$

PROPOSITIONAL LOGIC

- Khi biểu diễn tri thức bởi các công thức trong logic mệnh đề, cơ sở tri thức là một tập nào đó các công thức. Bằng phương pháp chuẩn hoá vừa nêu, thì cơ sở tri thức là một tập hợp các câu phức hợp (clause)

PROPOSITIONAL LOGIC

- Như vậy, mọi công thức đều có thể đưa về dạng chuẩn tắc là hội của các clause. Mỗi clause có dạng:

$$\text{Not}(A_1) \vee \dots \vee \text{Not}(A_m) \vee B_1 \vee \dots \vee B_n$$

trong đó A_i, B_i là các mệnh đề (literal dương).

- Câu Kowalski có dạng:

$$A_1 \wedge \dots \wedge A_m \rightarrow B_1 \vee \dots \vee B_n$$

PROPOSITIONAL LOGIC

- Khi $n \leq 1$, clause chỉ chứa nhiều nhất một literal dương. Ta gọi những câu như thế là câu Horn (Alfred Horn, 1951).
- Nếu $m > 0$, $n = 1$, câu Horn có dạng: $A_1 \wedge \dots \wedge A_m \rightarrow B$ trong đó các A_i được gọi là các điều kiện, còn B là kết luận.
- Các câu Horn còn gọi là các luật if-then.

PROPOSITIONAL LOGIC

- Khi $m = 0$ và $n = 1$, câu Horn trở thành các câu đơn.
- Không phải mọi công thức đều có thể biểu diễn dưới dạng hội các câu Horn.
- Trong các ứng dụng, cơ sở tri thức thường là tập hợp các câu Horn (tập hợp các luận if-then)

PROPOSITIONAL LOGIC

Luật suy diễn:

- Một công thức H được xem là hệ quả logic (logical consequence) của một tập các công thức $G = \{G_1, \dots, G_n\}$ nếu trong bất kỳ minh họa nào mà $\{G_1, \dots, G_m\}$ đúng thì H cũng đúng.
- Luật suy diễn là phương pháp sử dụng những tri thức có sẵn trong cơ sở tri thức để suy ra tri thức mới là hệ quả logic của các công thức đó.

PROPOSITIONAL LOGIC

- Luật Modus Ponens: $(A \rightarrow B, A) \Rightarrow B$
- Luật Mondus Tollens: $(A \rightarrow B, \text{not}(B)) \Rightarrow \text{not}(A)$
- Luật bắc cầu
- Luật loại bỏ hội

PROPOSITIONAL LOGIC

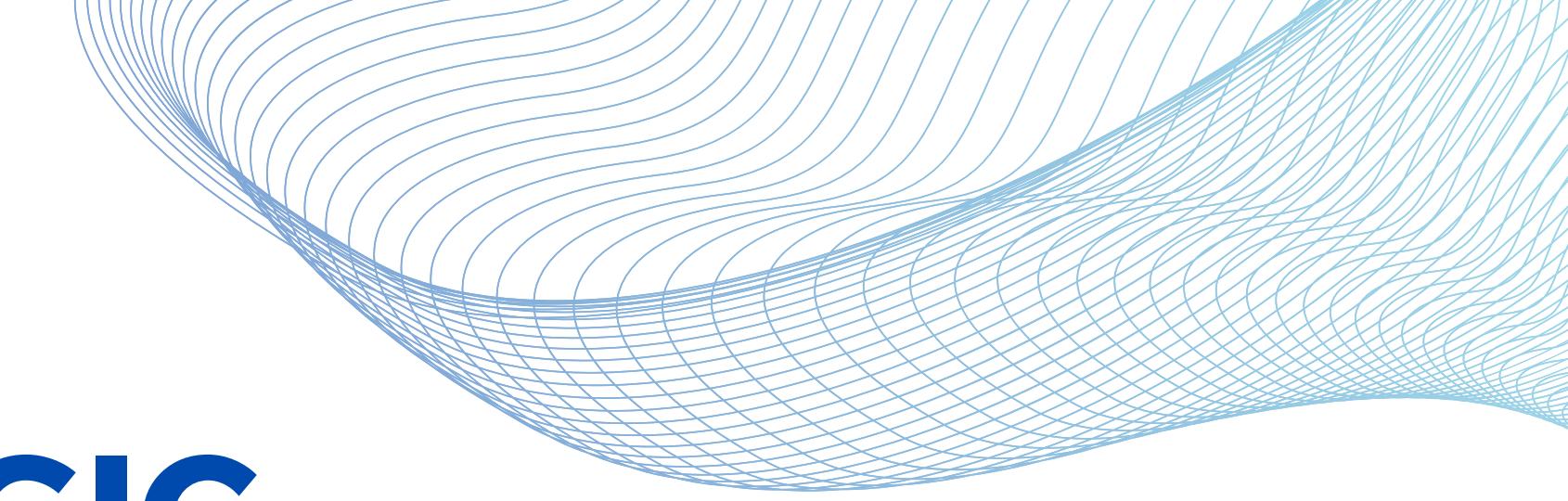
- Luật đưa vào hội
- Luật đưa vào clause
- Luật phân giải: $(A \vee B, \text{not}(B) \vee C) \Rightarrow (A \vee C)$
- Một luật suy diễn là tin cậy nếu bất kỳ mô hình nào của giả thiết của luật cũng là mô hình của kết luận.
- Chứng minh: luật phân giải là luật suy diễn tổng quát, bao gồm luật **Modus Ponens, Modus Tollens, luật bắc cầu (Homework)**

PROPOSITIONAL LOGIC

- Giả sử chúng ta có một tập các công thức. Bằng các luật suy diễn, ta có thể suy ra những công thức mới.
- Các công thức đã được cho được gọi là các tiên đề.
- Các công thức được suy ra được gọi là các định lý.
- Dãy các luật suy diễn được áp dụng để dẫn đến các định lý được gọi là một chứng minh của định lý

PROPOSITIONAL LOGIC

- Nếu các luật suy diễn là tin cậy, thì các định lý là hệ quả logic của các tiên đề.
- Trong các hệ tri thức, bằng cách sử dụng các luật suy diễn, người ta thiết kế lên các thủ tục suy diễn để từ các tri thức trong cơ sở tri thức, ta suy ra các tri thức mới đáp ứng nhu cầu người sử dụng



PROPOSITIONAL LOGIC

- **Hệ hình thức** (formal system) bao gồm một tập các tiên đề và một tập các luật suy diễn nào đó trong một ngôn ngữ biểu diễn tri thức nào đó.
- Một tập suy diễn được gọi là đầy đủ nếu mọi hệ quả logic của một tập các tiên đề đều chứng minh được bằng cách chỉ sử dụng các luật trong tập đó



PROOF BY CONTRADICTION

- Phương pháp chứng minh bác bỏ là phương pháp được sử dụng phổ biến trong toán học.
- Chứng minh bác bỏ bằng luật phân giải:
 - Luật phân giải trên các clause.
 - Luật phân giải trên các câu Horn.
 - Thuật toán Havard (1970).
 - Thuật toán Robinson (1971)

PROOF BY CONTRADICTION

- Để chứng minh P đúng, ta giả sử P sai và dẫn đến một mâu thuẫn.
- Để thuận tiện hơn cho việc sử dụng luật phân giải, ta sẽ cụ thể hóa luật phân giải trên các câu quan trọng



PROOF BY CONTRADICTION

- Luật phân giải trên các câu tuyên (clause)

$$\frac{\begin{array}{c} A_1 \vee \dots \vee A_m \vee C \\ | \\ \neg C \vee B_1 \vee \dots \vee B_n \end{array}}{A_1 \vee \dots \vee A_m \vee B_1 \vee \dots \vee B_n}$$

trong đó $A_1, A_2, \dots, A_m, C, B_1, \dots, B_n$ là literals.



PROOF BY CONTRADICTION

- Luật phân giải trên các câu Horn:

$$P_1 \wedge \dots \wedge P_m \wedge S \Rightarrow Q,$$

$$R_1 \wedge \dots \wedge R_n \Rightarrow S$$

$$P_1 \wedge \dots \wedge P_m \wedge R_1 \wedge \dots \wedge R_n \Rightarrow Q$$

- Hai câu có thể áp dụng được luật phân giải được gọi là hai câu phân giải được và kết quả nhận được gọi là phân giải thức của chúng

PROOF BY CONTRADICTION

- Hai câu phân giải được nếu một câu chứa một literal đối lập với một literal trong câu kia.
- Giả sử G là một tập các câu tuyển (clause). Ta ký hiệu $R(G)$ là tập bao gồm các câu thuộc G và tất cả các câu nhận được từ G thông qua dãy áp dụng luật phân giải.

PROOF BY CONTRADICTION

- Luật phân giải là luật đầy đủ để chứng minh một tập câu là không thoả được.
- Định lý phân giải: Một tập câu tuyển là không thoả được khi và chỉ khi $R(G)$ chứa câu rỗng



PROOF BY CONTRADICTION

Procedure Resolution

Input: G, tập các câu tuyễn

- Repeat
 - Chọn 2 câu A,B thuộc G.
 - Nếu A, B phân giải được tính Res(A,B). Nếu Res(A,B) là câu mới, thêm vào G.

until nhận được [] hoặc ko có câu mới xuất hiện

- if nhận được câu rỗng then G không thoả được else G thoả được.

PROOF BY CONTRADICTION

- Định lý phân giải có nghĩa là nếu từ các câu thuộc G, bằng cách áp dụng luật phân giải, ta dẫn đến câu rỗng thì G là không thoả được, còn nếu không thể sinh ra câu rỗng bằng luật phân giải thì G thoả được.
- Việc dẫn đến câu rỗng tức là đã dẫn đến hai literal đối lập nhau (hay dẫn đến mâu thuẫn) được.



PROOF BY CONTRADICTION

- Ví dụ: giả sử G là tập hợp các câu tuyển sau:

$$\neg A \vee \neg B \vee P \quad (1)$$

$$\neg C \vee \neg D \vee P \quad (2)$$

$$\neg E \vee C \quad (3)$$

$$A \quad (4)$$

$$E \quad (5)$$

$$D \quad (6)$$

Chứng minh P là hệ quả logic của các câu trên.

PROOF BY CONTRADICTION

- Thông thường chúng ta sẽ sử dụng bảng chân lý để kiểm tra tính đúng đắn của một biểu thức.
- Ngoài ra, chúng ta có thể sử dụng hai thuật toán sau đây:
 - Thuật toán Harvard (1970)
 - Thuật toán Robinson (1971)

HARVARD ALGORITHM

- Step 1: phát biểu lại giả thiết (GT) và kết luận (KL) của bài toán dưới dạng chuẩn sau:

$$GT_1, \dots, GT_n \rightarrow KL_1, \dots, KL_m$$

trong đó, GT_i , KL_j được xây dựng từ các biến mệnh đề và các phép nối: AND, OR, NOT.

- Step 2: bước bỏ phủ định. Khi cần bỏ các phủ định, chuyển về GT_i sang về kết luận KL_j và ngược lại.
- Step 3: Thay dấu AND ở GT_i và OR ở KL_j bằng “,”

HARVARD ALGORITHM

- Step 4: Nếu GT_i còn dấu OR và KL_j còn dấu AND, tách chúng thành hai dòng con.
- Step 5: Một dòng được chứng minh nếu tồn tại chung một mệnh đề ở cả hai vế.
- Step 6: Bài toán được chứng minh khi và chỉ khi các dòng được chứng minh. Ngược lại, bài toán không được chứng minh.

ROBINSON ALGORITHM

Robinson đã cải tiến thuật toán Harvard.

- Step 1: phát biểu lại giả thiết (GT) và kết luận (KL) của bài toán dưới dạng chuẩn sau:

$$GT_1, \dots, GT_n \rightarrow KL_1, \dots, KL_m$$

trong đó, GT_i , KL_j được xây dựng từ các biến mệnh đề và các phép nối: AND, OR, NOT.

- Step 2: Thay dấu AND ở GT_i và OR ở KL_j bằng “,”

ROBINSON ALGORITHM

- Step 3: Chuyển về KL_j sang vẽ GT_i với dấu phủ định để còn một vẽ.
- Step 4: Xây dựng một mệnh đề mới bằng cách tuyển một cặp mệnh đề từ danh sách các mệnh đề. Nếu mệnh đề mới có các biến mệnh đề đối ngẫu thì mệnh đề đó được loại bỏ.
- Step 5: bổ sung mệnh đề mới này vào danh sách và lặp lại bước 4.
- Step 6: bài toán được chứng minh khi và chỉ khi còn hai mệnh đề đối ngẫu. Ngược lại bài toán không được chứng minh