Artificial Intelligence

academic year 2025/2026

Giorgio Fumera

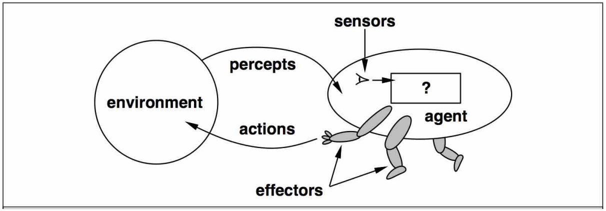
Pattern Recognition and Applications Lab
University of Cagliari (Italy)

Solving Problems by Searching

Suggested textbook

S. Russell, P. Norvig, *Artificial Intelligence – A Modern Approach*, 4th Ed., Pearson, 2021 (or a previous edition)

The *rational agent* approach to AI

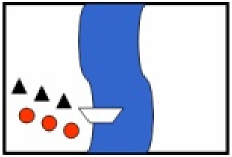


Systems that **act rationally**, according to a **well-defined objective** – regardless of whether they are “intelligent” or not.

Example problem: missionaries and cannibals

Assume your goal is to design a **rational agent**, in the form of a computer program, capable of **autonomously** solving the following problems.

A classic AI toy-problem: three missionaries and three cannibals must cross a river on a boat that can only hold two people, without leaving more cannibals than missionaries on either side of the river. How can all six get across the river safely, possibly with the **minimum** number of crossings?



Example problem: 15-puzzle

Another classic AI's toy-problem: transform an array of tiles from an initial configuration into a desired one, by a sequence of moves of a tile into an adjacent empty cell (possibly, the **shortest** sequence).

An example:

13	10	11	6
5	7	4	8
1		14	9
3	15	2	12

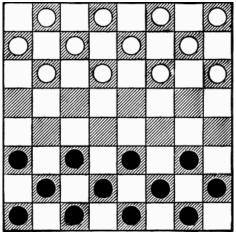
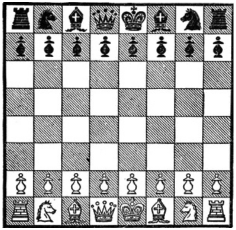
initial configuration

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

desired configuration

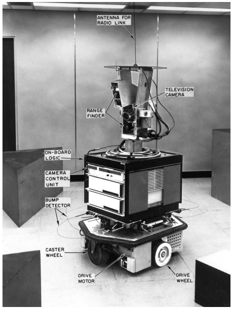
Example problems: draughts and chess

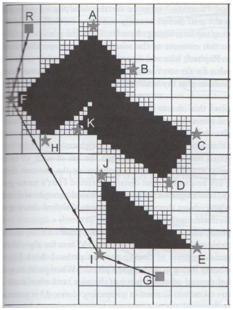
Two historical problems addressed by many researchers since the early days of AI (chess has been named the “Drosophila of AI”).

Example problems: robot navigation

A real-world problem addressed since mid–1960s.

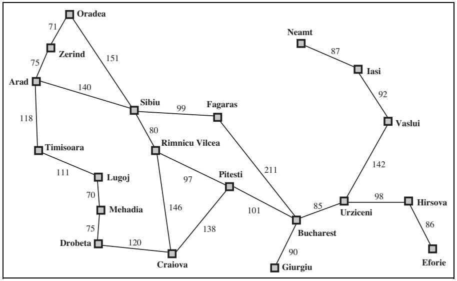




Left: Shakey the robot (1968). Right: formalisation of a navigation problem for Shakey: the shortest route from any two points R and G, avoiding obstacles (black polygons), can be found considering only straight line segments connecting the polygons' vertices.

Example problems: route finding in maps

Find a route (possibly, the **shortest** one) from any two cities in a map, e.g., from Arad to Bucharest.



A framework for search problems

Problems like the previous ones can be formalized as follows:

1. **goal formulation**: what are the desired “world **states**”?
2. **problem formulation**
   - what **actions** to consider?
   - what **states** to consider?
   - what is the **initial state**?

   crucial point: finding a proper level of **abstraction**, by removing irrelevant details

Under the above formulation:

- **solution** of a search problem: a **sequence of actions** from the initial state to the goal state
- **search**: the process of looking for a solution

Goal and problem formulation: examples

15-puzzle

13	10	11	6
5	7	4	8
1		14	9
3	15	2	12

initial configuration

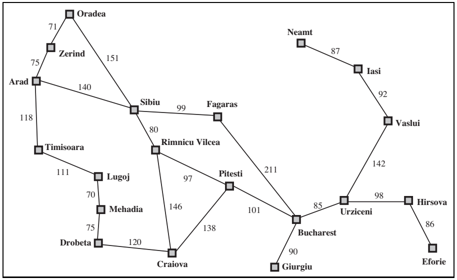
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

desired configuration

- **goal**: getting from the initial to the desired tile configuration (possibly, by the shortest sequence of moves)
- **states**: each possible 16! tile configurations
- **actions**: moving to the empty cell any of the tiles (two to four) adjacent to it

Goal and problem formulation: examples

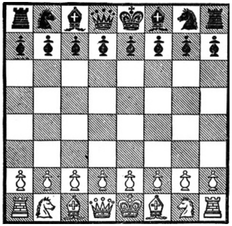
Route finding in maps



- **goal**: getting from the starting city to the destination one (possibly, through the shortest route)
- **states**: (being in) each possible city
- **actions**: moving between two **adjacent** cities

Goal and problem formulation: examples

Chess



- **goal**: to checkmate – a **precise** description is definitely better than the enumeration of **all** possible chessboard configurations
- **states**: each legal chessboard configuration
- **actions**: all legal moves of a single piece from any given legal chessboard configuration

Properties of search problems

- **Static** vs **dynamic**: does the environment change over time? Examples: 15-puzzle and chess are static; robot navigation is dynamic, if the position of obstacles changes over time
- Fully vs partially **observable**: is the current state completely known? Examples: 15-puzzle and chess are fully observable; robot navigation is partially observable, if sensors are not "perfect"
- **Discrete** vs **continuous** sets of states and actions. Examples: 15-puzzle and chess are discrete, robot navigation is continuous
- **Deterministic** vs **non-deterministic**: is the outcome (the resulting state) of any sequence of actions certain, i.e., known in advance? Examples: 15-puzzle is deterministic, chess is not (due to the opponent's move, that is unknown when deciding one's own)

Examples of real-world search problems

Many challenging real-world problems can be formulated as search problems. Some examples:

- **traveling salesperson problem**: finding the **shortest** tour that allows one to visit every city of a given map **exactly once** (applications to planning, logistics, microchip manufacture, DNA sequencing, etc.)
- **route-finding**: routing in computer networks, airline travel planning, etc.
- **VLSI design**: cell layout, channel routing

Search problems: formal defintion

How to devise algorithms, and how to implement them using some programming language, to solve search problems?

First, a rigorous **problem definition** is needed.

The goal and problem formulation sketched above can be formally defined in terms of four components:

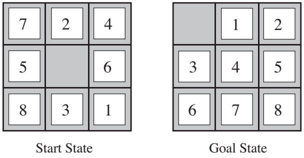
- the **initial state**
- the set of possible **actions**
- the **goal test**
- the **path cost**

Search problems: formal defintion

- **Initial state**: a description of the state where the search starts
- **Actions**: a description of all possible actions available at any possible state. They can be defined as a **successor function** SF which, given a state s, returns the set of ordered pairs (a', s'), where a' is a legal action in state s and s' the resulting state
  - the initial state and SF **implicitly** define the **state space**: a **graph** whose nodes correspond to states and edges to actions
  - a **path** in the state space is a sequence of states connected by a sequence of actions
- **Goal test**: a function that determines whether or not any given state is a goal state
- **Path cost**: the cost of a path, **depending on the problem's goal**. It is often given by the sum of the costs of the individual actions (**step cost**) along the path

Example: 8-puzzle

A simpler version of the 15-puzzle problem – an example:



- **States**: all possible 9! board configurations
- **Initial state**: any given board configuration
- **Goal test**: is a given state equal to the goal state?
- **Path cost**: if the goal is to reach the goal state by the **shortest** sequence of moves, each action "costs" 1 move, and the path cost is the number of steps (moves) in the path

(cont.)

Example: 8-puzzle

- **Actions**: a convenient description:
  - switching the **blank** (i.e., the empty position, considered as a fictitious tile) with any of its (two to four) adjacent tiles
  - this corresponds to considering **four** possible actions, which can be named *up*, *down*, *left*, *right*
  - a solution (action sequence) would look like (*left*, *up*, *up*, . . .)

Implementation example in Python:

- states can be represented by tuples, using `0` for the blank, e.g., the start state shown above:

```
(7, 2, 4, 5, 0, 6, 8, 3, 1)
```

- actions can be represented as strings, e.g., `"up"`, etc.
- *SF*(`<state>`) should return all the states reachable from `<state>` with the corresponding actions; e.g., for the start state above, *SF* should return a set of four state/action tuples including `((7, 2, 4, 5, 3, 6, 8, 0, 1), "down")`

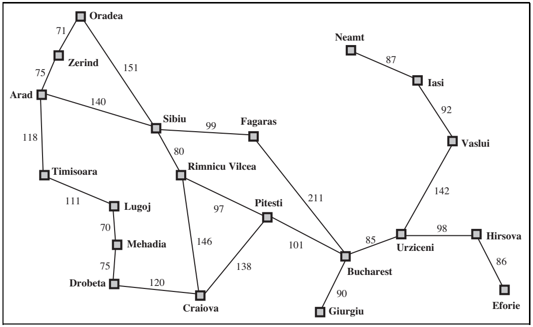
Example: 8-puzzle

Some notes on the 8-puzzle problem:

- the space state is very large (9! nodes); fortunately, it is not necessary to store it into computer memory: it is **implicitly** defined by the initial state and *SF*
- for 8-puzzle (and 15-puzzle) only **half** of the configurations can be reached by any given starting state (Woolsey Johnson and Story, 1879 – `https://doi.org/10.2307/2369492`)
  - the resulting state space is a graph of 9! nodes, partitioned into two **disconnected** graphs of $\frac{9!}{2}$ nodes each
  - therefore, only half of all possible 8-puzzle (and 15-puzzle) instances are solvable
  - checking for solvability is easy, e.g., see the paper by Woolsey Johnson and Story

Example: route finding in maps

A very simple problem setting: a set of cities, with roads directly connecting pairs of cities (i.e., no road junctions), where only route length matters (i.e., the objective can only be to find a route between any two cities, usually the shortest one) – an example:



Example: route finding in maps

- **States**: the set of cities in the map
- **Initial state**: any given city in the map
- **Goal test**: is a given state (city) equal to the goal state (destination city)?
- **Path cost**: if the goal is to find the **shortest** route, the path cost is route length, which equals the sum of the lengths of all roads connecting adjacent cities in the route

(cont.)

Example: route finding in maps

- **Actions**: a convenient choice:
  - moving from any given city to any **adjacent** one
  - in a map with n roads directly connecting pairs of cities, this amounts to $2n$ possible actions (provided there are no one-way roads), e.g., in the previous map: Arad $\rightarrow$ Zerind
  - a solution, e.g., from Arad to Bucharest, would be a sequence of adjacent cities, e.g.: (Arad, Sibiu, Fagaras, Bucharest)
  - no **explicit** description of actions is needed: they can be easily determined from a given path, e.g.:
    Arad $\rightarrow$ Sibiu, Sibiu $\rightarrow$ Fagaras, Fagaras $\rightarrow$ Bucharest

Implementation example in Python:

- states can be represented as strings, e.g., `"Arad"`
- SF(`<state>`) should return all the cities adjacent to `<state>`, e.g., as a tuple:
  $SF(\texttt{"Arad"}) = (\texttt{"Timisoara"}, \texttt{"Sibiu"}, \texttt{"Zerind"})$

Example: route finding in maps

Some notes on the considered kind of route finding problem:

- it is easy to see that the state space corresponds to the map, considered as a graph
- contrary to the 8-puzzle problem, in this case the **whole** state space must be stored in computer memory beforehand, since it encodes all the **necessary** information to solve the problem (i.e., states, available actions and their cost)

For instance, the map can be represented in Python as a dictionary with city names as keys and nested dictionaries as values, made up in turn of adjacent cities as keys and the corresponding road lengths as values, e.g.:

```
{"Arad": {"Timisoara": 118, "Sibiu": 140, "Zerind": 75},
 "Zerind": {"Arad": 75, "Oradea": 71},
 ... }
```

Solving a search problem

This course focuses on the simplest kind of search problem: **static**, **fully observable**, **discrete**, **deterministic** – examples:

- 8-puzzle
- route finding in maps (under the setting considered above)
- Rubik's cube

Key property: a solution can be searched for **offline**, i.e., before executing any action.

Main steps for solving this kind of search problem:

1. goal and problem **formulation** (discussed above)
2. **searching** for a solution (to be discussed in the following)
3. **executing** the corresponding actions (not considered in this course)

Search algorithms

A possible approach for solving a search problem: systematically computing **all** the possible sequences of actions (i.e., all the paths in the state space) from the initial state, until a goal state is reached.

Procedure: **iteratively** constructing action sequences from the initial state, by **expanding** at each iteration **one** of the current sequences, i.e., adding to it **every** possible **single** action.

During this process, the current set of action sequences can be conveniently represented by a **tree graph**, named **search tree**.

The criterion used for choosing **which** action sequence to expand at each iteration defines a **search strategy**.

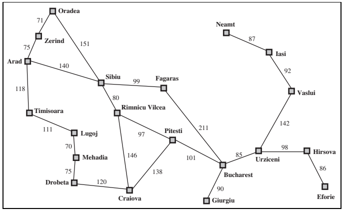
Search tree

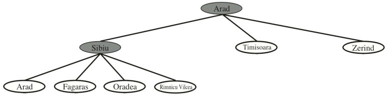
A **search tree** represents a set of action sequences (**partial solutions**) starting from the initial state:

- each **node** represents a state
- an **edge** represents the action that leads from the parent node's state to the child node's state
- a **leaf node** corresponds to the end state of an action sequence
- each **path** (sequence of connected nodes) from the root to a leaf represents a **distinct** sequence of actions and of the resulting sates
- the **depth** of a node is the number of actions in the path from the root to that node (the root node has **zero** depth)
- the set of **leaf nodes** is called **fringe** or **frontier**

Note that a **same** state can appear in **different** nodes of a search tree, i.e., it can be part of **different** paths.

Search tree: an example

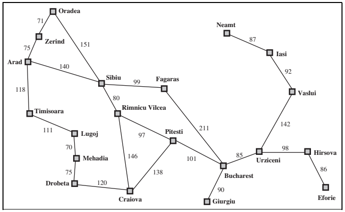


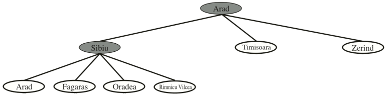


Goal: finding a route from Arad to Bucharest – a possible, **partial** search tree is shown on the right

- root node: the start city, Arad
- six leaf nodes (fringe, in white): six **distinct**, partial solutions, e.g.:
  - Arad $\rightarrow$ Sibiu $\rightarrow$ Arad
  - Arad $\rightarrow$ Sibiu $\rightarrow$ Fagaras
  - ...

State space and search tree





Note that a search tree is **different** from the state space (graph).

This is evident in the considered route finding problem, where the map corresponds to the state space.

Sketch of a general tree-search algorithm

1. construct the root node R of the search tree, associate the initial state to R, and set the fringe equal to {R} (the initial state is the only leaf, or partial solution, at this point)
2. repeat:
    - 2.1 if the fringe is empty, then no solution has been found and the algorithm stops
    - 2.2 choose **one** leaf node (partial solution) N from the fringe
    - 2.3 if N contains a goal state, then the search is successfully completed: the algorithm returns the sequence of actions in the path from R to N as a solution
    - 2.4 **expand** the state in N:
        - 2.4.1 apply *SF* to the state in N
        - 2.4.2 for each state **generated** by *SF*, construct a new leaf node, add it to the tree as a child of N, and add it to the fringe
        - 2.4.3 remove N from the fringe

Sketch of a general tree-search algorithm

The tree-search algorithm is **independent** of the search problem.

The key point is the choice of a leaf node to **expand** in step 2.2:

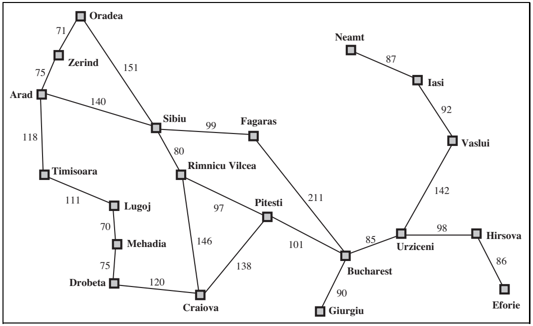
- **different** criteria can be used for this choice
- each criterion defines a specific **search strategy**
- each search strategy leads to a different **search algorithm**

It turns out that different search strategies can have very different **performance**, in terms of:

- **effectiveness**: the quality of the solution found (not necessarily the best – minimum cost – one)
- **efficiency**: the amont of computing resources required to find a solution (processing time and memory requirements)

Tree-search algorithm: an example

Route finding in maps: getting from Arad to Bucharest



Tree-search algorithm: an example

Step 1: the root node is **generated**, corresponding to the initial state (Arad).

(a) The initial state

Tree-search algorithm: an example

Step 2: the first iteration starts.

Step 2.1: the fringe is not empty

Step 2.2: the fringe contains a single leaf node, the root node (Arad), which is therefore selected

Step 2.3 (goal test): Arad is not the goal state

Step 2.4: the chosen leaf (the root node) has to be **expanded**

(a) The initial state

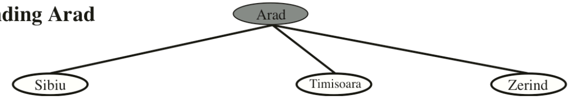
Tree-search algorithm: an example

Step 2.4.1: the successor function is applied to the state Arad, which **generates** **all** the states (cities) adjacent to Arad

Step 2.4.2: the newly generated states are added as child nodes to the root node, and to the fringe

Step 2.4.3: the expanded node is removed from the fringe

(b) After expanding Arad



Leaf nodes (shown in white) are by definition not yet expanded; non-leaf nodes (shaded) are the ones already expanded.

Tree-search algorithm: an example

Step 2: a new iteration starts.

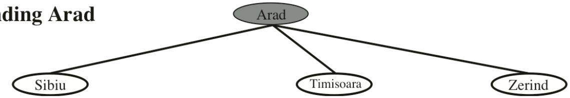
Step 2.1: the fringe is not empty

Step 2.2: the fringe contains three leaf nodes: a criterion for choosing one should be used – for now, assume that the leaf corresponding to Sibiu is chosen

Step 2.3 (goal test): Sibiu is not the goal state

Step 2.4: the chosen leaf node has to be **expanded**

(b) After expanding Arad



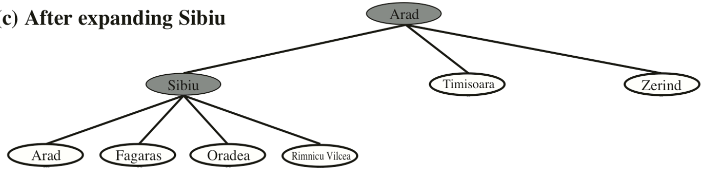
Tree-search algorithm: an example

Step 2.4.1: the successor function is applied to the state Sibiu, which **generates** **all** the states (cities) adjacent to Sibiu

Step 2.4.2: the newly generated states are added as child nodes to the expanded node, and to the fringe

Step 2.4.3: the expanded node is removed from the fringe

(c) After expanding Sibiu

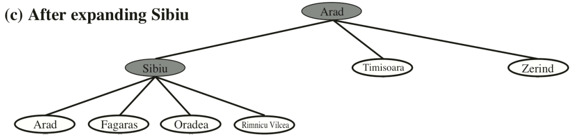


Then a new iteration starts . . .

Tree-search algorithm: an example

The current search tree contains **six** action sequences (partial solutions), corresponding to the paths from the root to each of the **six** leaf nodes:

1. Arad $\rightarrow$ Sibiu $\rightarrow$ Arad
2. Arad $\rightarrow$ Sibiu $\rightarrow$ Fagaras
3. Arad $\rightarrow$ Sibiu $\rightarrow$ Oradea
4. Arad $\rightarrow$ Sibiu $\rightarrow$ Rimnicu Vilcea
5. Arad $\rightarrow$ Timisoara
6. Arad $\rightarrow$ Zerind



General tree-search algorithm

A more concise (but still informal) description, as a function with two abstract arguments

- `problem`: a data structure with information on the search problem at hand (initial state, goal test, successor function, etc.)
- `strategy`: a criterion for choosing the next leaf to expand

```
function TREE-SEARCH (problem, strategy)
returns a solution, or failure
    generate the root node using the initial state of problem
    loop do
        if there are no leaf nodes
            then return failure
        choose a leaf node according to strategy
        if the chosen leaf node contains a goal state
            then return the corresponding solution
        expand the chosen leaf node
```

Implementation hints

In the following, a more formal version of the above tree-search algorithm is presented in the form of pseudo-code, together with the corresponding data structures.

Note that this version is **independent** of the search problem and of any programming language.

Details may change depending on the specific programming language used for implementation.

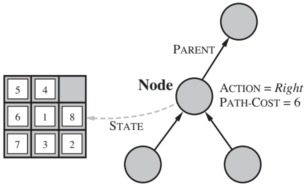
Data structures: nodes of the search tree

Information to be stored in computer memory to represent a single **node** of the search tree:

- the **state** associated to the node
- a reference to the **parent** node, to reconstruct the **path** from the root when a goal state is found
- the **action** that lead to this node from the parent one
- the **path cost** from the root node to this one

Additional information may be useful, e.g., the node **depth**.

An example for the 8-puzzle problem:



Data structures: nodes of the search tree

A **record** data structure, like the C language's `struct`, a dictionary in Python, or instance variables of a **class** in object–oriented languages, can be used, including the following fields:

STATE	a problem-dependent representation of the corresponding state, e.g., the tuple `(5, 4, 0, 6, 1, 8, 7, 3, 2)`
PARENT-NODE	a reference (e.g., in Python) or pointer (e.g., in C) to the parent node
ACTION	a description of the action executed from the parent node (e.g., the string "Right")
PATH-COST	the total cost of the actions on the path from the root to this node
DEPTH	the number of actions in the path from the root to this node

Data structures: fringe of the search tree

At each step of the tree-search algorithm one of the **leaf** nodes (i.e., an element of the fringe) is selected for expansion.

Leaf nodes must therefore be **quickly** accessible.

To this aim it is convenient to store into a **linear** data structure (e.g., a **linked list** in C or a **list** in Python) references or pointers to leaf nodes.

The fringe can be implemented as a **queue**, a **first-in first-out** (FIFO) data structure

- newly generated nodes must be inserted into the queue in the order in which they will be chosen by the search strategy at hand
- this way, the next leaf node to expand is by definition the **first** one in the current fringe

Data structures: the search problem

Information specific to the search problem at hand (the `problem` argument) can be stored into a **distinct** record data structure:

INITIAL-STATE	a problem-dependent representation of the initial state
GOAL-TEST	a function that checks whether a given state is a goal state
SUCCESSOR-FN	the function *SF* (see above) that returns a set of pairs (*state*, *action*) from a given state
STEP-COST	a function that returns the cost of carrying out a given action from a given state

The values of the GOAL-TEST, SUCCESSOR-FN and STEP-COST fields can be **pointers to functions** in a C `struct` or **references** to functions (i.e., function names) in a Python dictionary; these functions can also be implemented as class methods in object-oriented programming.

Implementation of the tree-search algorithm

A **possible** pseudo-code for the tree-search algorithm, with SMALL CAPITALS denoting function and field names, and FIELD-NAME[record] denoting the value of the field FIELD-NAME of a `record`:

```
function TREE-SEARCH (problem, ENQUEUE)
returns a solution, or failure
  fringe ← an empty queue
  fringe ← ENQUEUE(MAKE-NODE(INITIAL-STATE[problem]),
                           fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem](STATE[node]) succeeds
    then return SOLUTION(node)
    fringe ← ENQUEUE(EXPAND(node, problem), fringe)
```

Implementation of the tree-search algorithm

Note that the above implementation is **problem-independent**, and is an example of **modular** programming style:

- it can be used for **any** search problem
- all **problem-specific** details (e.g., the data structure representing a state and the goal-test function) are represented or implemented **separately** from data structures and functions related to the search tree

The search strategy is assumed to be defined through the function ENQUEUE, which is passed to TREE-SEARCH as an argument (e.g., a pointer to a function in C language).

As suggested above, ENQUEUE inserts the newly generated nodes into a queue, i.e., in the order in which they have to be expanded according to the corresponding search strategy: therefore the **first** leaf node in the current fringe is always selected for expansion.

Implementing auxiliary functions: node expansion

This function expands a node, connects its children (leaf) nodes to it, and returns them:

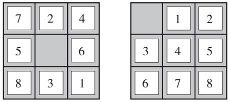
```
function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each ⟨action, result⟩ in
            SUCCESSOR-FN[problem](STATE[node]) do
        n ← MAKE-NODE(result)
        PARENT-NODE[n] ← node
        ACTION[n] ← action
        PATH-COST[n] ← PATH-COST[node] +
                        STEP-COST[problem](node, action)
        DEPTH[n] ← DEPTH[node] + 1
        add n to successors
    return successors
```

Other auxiliary functions

- MAKE-NODE(`s`) returns a new instance of the node data structure, storing the state `s` in its STATE field, with no parent, no action, zero depth and zero path-cost
- REMOVE-FIRST(`q`) removes the first element from the queue `q` and returns it
- SOLUTION(`n`) returns the sequence of actions (the values of the ACTION fields) from the root of the tree to node `n`, following the pointers in the PARENT-NODE fields from `n` backwards to the root
- ENQUEUE(`nodes`, `q`) inserts in the queue `q` each node in the set `nodes`, in a position defined by the **search strategy**. Accordingly, a **different** implementation of ENQUEUE must be defined for each search strategy

A note on the representation of the state space

For some search problems the state space can be very large.
For instance, the state space of the 8-puzzle game has size 9!.

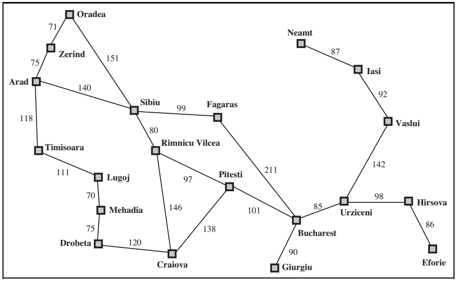


Start State   Goal State

Nevertheless, as mentioned above, for problems like 8-puzzle it is not necessary to store the whole state space in memory: the initial state and the successor function *SF* are sufficient to build any search tree. In other words, they **implicitly** define the state space.

A note on the representation of the state space

On the other hand, in problems like route finding in maps the whole state space (e.g., the graph corresponding to the map) need to be stored **explicitly** into computer memory beforehand (e.g., as nested dictionaries in Python, as suggested above), to be used by the successor function *SF*.



Measuring problem-solving performance

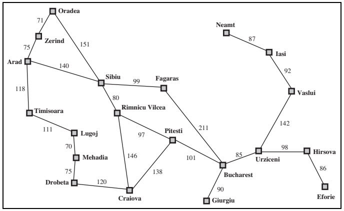
The performance of a tree-search algorithm can be evaluated according to two main criteria:

- **effectiveness**: how “good” is the solution found (if any)?
  - **completeness**: is the algorithm **guaranteed** to find a solution, when there is one?
  - **optimality**: when a solution is found, is it the **best** one, i.e., is its path cost **minimal**?
- **efficiency**: what is the processing cost of finding a solution? Formally, this property is called **computational complexity**
  - **time** complexity: **how long** does it take to find a solution?
  - **space** complexity: **how much memory** is needed?

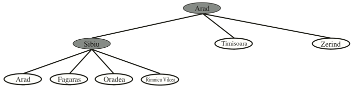
Often a **trade-off** between effectiveness and efficiency is required.

Search strategies

Search strategies differ **only** in the criterion to choose the leaf node (partial solution) to follow up at each step. An example:



current search tree:



which of the six partial solutions should one choose?

Two kinds of search strategies exist, depending on the available information about which choice is "better" than another:

- **uninformed** search strategies
- **informed** search strategies

Uninformed search strategies

In absence of any information about the “best” partial solution to select, the rationale of **uninformed** strategies is to **systematically** explore the space state, until a solution is found (if any).

Main uninformed strategies:

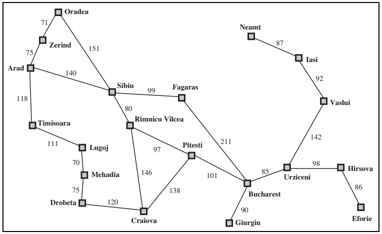
- breadth-first
- depth-first
- uniform-cost
- depth-limited
- iterative-deepening depth-first
- bidirectional

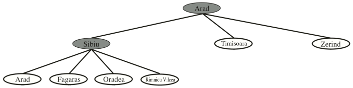
Breadth-first search (BFS)

BFS expands first the **shallowest** leaf node. If there is more than one leaf node at the lowest depth, one of them is **randomly** chosen.

This amounts to expand first all nodes at depth 0 (the root), then all nodes at depth 1, then all nodes at depth 2, and so on.

An example:

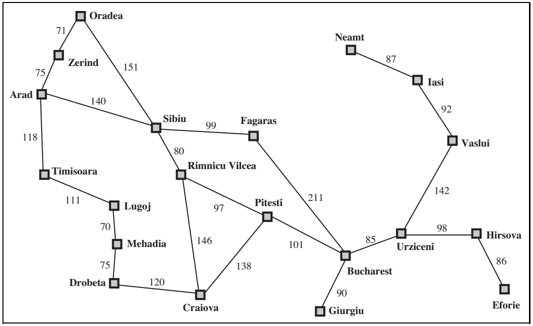




shallowest leaf nodes:
Timisoara, Zerind;
one of them is randomly chosen

Example of breadth-first search (1/6)

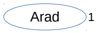
Route finding on maps: getting from Arad to Bucharest.



The nodes of the search tree will be numbered to to avoid ambiguities, since different nodes can be associated with the same state. The search steps are numbered according to the **general tree-search algorithm** shown above.

Example of breadth-first search (2/6)

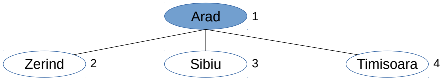
1. root node: the initial state, Arad;   fringe = (Arad (1))
2.1 the fringe is not empty
2.2 the **shallowest** node in the fringe (the root) is chosen (Arad)
2.3 Arad is not a goal state



Example of breadth-first search (3/6)

2.4 Arad is removed from the fringe and expanded, generating three nodes having **identical** depth

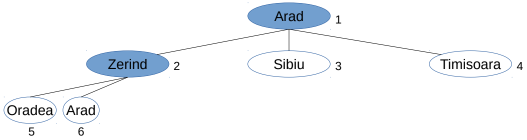
- the newly generated nodes must be inserted in the fringe (a queue) in the order in which they will be expanded by BFS; by definition, in BFS a newly expanded node has depth equal or higher than **all** the other nodes in the fringe; therefore, the newly expanded nodes are inserted **at the end** of the fringe
- since the newly generated nodes always have **identical** depth, they are inserted in the fringe in **any** order between themselves



Current fringe: (Zerind (2), Sibiu (3), Timisoara (4)).

Example of breadth-first search (4/6)

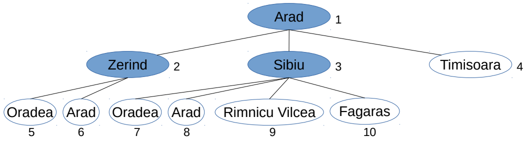
2.1 the fringe is not empty
2.2 the **first** node in the fringe is selected (it is guaranteed to be one of the shallowest leaf nodes)
2.3 the corresponding state, Zerind, is not the desired state
2.4 Zerind is removed from the fringe and expanded, generating two nodes that are inserted at the end of the fringe



Current fringe: (Sibiu (3), Timisoara (4), Oradea (5), Arad (6)).

Example of breadth-first search (5/6)

2.1 the fringe is not empty
2.2 the **first** node in the fringe is selected
2.3 the corresponding state, Sibiu, is not the desired state
2.4 Sibiu is removed from the fringe and expanded, generating four nodes that are inserted at the end of the fringe



Current fringe: (Timisoara (4), Oradea (5), Arad (6), Oradea (7), Arad (8), Rimnicu Vilcea (9), Fagaras (10)).

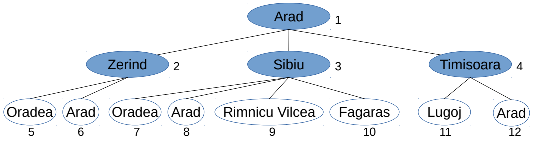
Example of breadth-first search (6/6)

2.1 the fringe is not empty

2.2 the **first** node in the fringe is selected

2.3 the corresponding state, Timisoara, is not the desired state

2.4 Timisoara is removed from the fringe and expanded, generating two nodes that are inserted at the end of the fringe

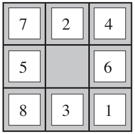


Current fringe: (Oradea (5), Arad (6), Oradea (7), Arad (8), Rimnicu Vilcea (9), Fagaras (10), Lugoj (11), Arad (12)).

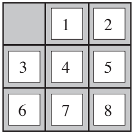
And so on.

Exercise

Apply the BFS algorithm to the 8-puzzle problem, considering the initial and goal states below, and **expanding** the first **four** nodes of the search tree (i.e., execute the first four iterations of the **general tree-search algorithm**).



Start State



Goal State

Properties of breadth-first search

In terms of **effectiveness**, it can be easily shown that BFS is:

- **complete**: a solution is always found, if one exists
- **non-optimal**: it is not guaranteed that the solution with minimum path cost is found (if any), unless the path cost is a non-decreasing function of depth

Computational complexity can be evaluated as follows.

Computational complexity of algorithms

The execution time of a given algorithm depends on several factors not intrinsic to the algorithm itself: its implementation in a specific language, the hardware on which the program is executed, etc.

Time complexity is therefore evaluated not in terms of the execution time, but in terms of the number of “elementary operations” carried out by the algorithm, assuming that each of them can be executed in constant time.

Depending on the algorithm, “elementary operations” can be additions, multiplications, comparisons, iterations, etc.

The first step is therefore to identify what the “elementary operations” of a given algorithm are. For instance:

- sorting algorithms (selection sort, quick sort, etc.): comparison between a pair of values
- computing the digits of the representation of a number in a given basis: computing the quotient and remainder of a division

Computational complexity of algorithms

The number of elementary operations carried out by an algorithm depends either on the **value** or the **size** of its input, e.g.:

- to sort a sequence of numbers, the number of comparisons depends on the **size** (length) of the sequence
- to computing the digits of the representation of a number in a given basis, the number of divisions depends on its **value**

Even if the number of elementary operations depends on more than one factor, for the sake of simplicity a single factor is considered and the others are kept constant.

Computational complexity of algorithms: an example

Consider the well-known **selection sort** algorithm, that sorts a sequence of values in increasing or decreasing order.

It can be described as follows:

1. find the minimum value in the sequence
2. swap it with the value in the first position
3. repeat the above steps for the remainder of the sequence, starting at the second position, then at the third one, up to the **penultimate** position

Computational complexity of algorithms: an example

A possible implementation of selection sort in C language, for sorting in non-decreasing order an array of integers:

```
void selection_sort (int a[], int length) {
   int i, j, ind_min, tmp;
   for (i = 0; i < length - 1; i++) {
      ind_min = i;
      for (j = i + 1; j < length; j++)
         if (a[j] < a[ind_min]) ind_min = j;
      if (ind_min != i) {
         tmp = a[i]; a[i] = a[ind_min]; a[ind_min] = tmp;
      }
   }
}
```

Computational complexity of algorithms: an example

Time complexity of selection sort can be evaluated as follows:

- each **comparison** in the nested loop can be considered as an **elementary operation**
- for a sequence of length n, the outer loop is repeated for $n-1$ times
- at iteration k of the outer loop ($k = 1, \ldots, n-1$), $n-k$ comparisons are made to find the lowest element, starting from the k-th position, then a swap is possibly made.

The **exact** number of comparisons is therefore given by:

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Time complexity depends therefore on the **size** of the input, i.e., the sequence length n. Therefore, all problem instances of a given size (sequences of a given length) have the same time complexity.

Computational complexity of algorithms

Let n denote the value of the factor which the number of elementary operations carried out by an algorithm depends on (e.g., the length of a sequence to be sorted, or the value of a number to be represented in a given basis), and $f(n)$ the corresponding number of of elementary operations.

For some algorithms evaluating $f(n)$ can be difficult, especially if it depends on the problem **instance**, i.e., on the input value.

Time complexity is thus evaluated only for some particular cases, usually:

- **worst-case** complexity, corresponding to the instances that require the **highest** number of elementary operations
- **average-case** complexity: average number of elementary operations over **all** possible problem istances
- **best-case** complexity, corresponding to the instances that require the **lowest** number of elementary operations

Computational complexity of algorithms

To evaluate and compare algorithms it useful to consider their **asymptotic** time complexity, i.e., the behaviour of $f(n)$ as $n \to \infty$.

To this aim an **upper bound** $g(n)$ of $f(n)$ is determined using asymptotic analysis, known as "big 'O' notation".

A function $f(n)$ is said to be $\mathcal{O}(g(n))$ ("order g"), if there exists some $n_0 > 0$ and $c > 0$ such that $f(n) \leq c \times g(n)$ for each $n \geq n_0$. Of course, the **tightest** upper bound $g(n)$ is of interest.

For instance, polynomials of degree p are $\mathcal{O}(n^p)$:

$$a_p n^p + a_{p-1} n^{p-1} + \ldots + a_1 p + a_0$$

As an example, this implies that the time complexity of selection sort, given by $\frac{n(n-1)}{2}$, is $\mathcal{O}(n^2)$.

Computational complexity of algorithms

Well-known categories of (increasing) asymptotic complexity are the following:

- $\mathcal{O}(1)$: constant time algorithms: their execution time is identical for all problem instances
- $\mathcal{O}(log\ n)$: logarithmic time (e.g., binary search in sorted sequences)
- $\mathcal{O}(n)$: linear time (e.g., sequential search)
- $\mathcal{O}(n^p)$, for a given integer p: polynomial time (e.g., selection sort, with $p = 2$)
- $\mathcal{O}(k^n)$, for a given $k > 1$: exponential time (e.g., the simplex algorithm in linear programming)

Computational complexity of search algorithms

Getting back to the **general tree-search algorithm**:

- elementary operation: **generation of a new node** during the expansion of a leaf node (step 2.4)
- data to be stored in memory: **nodes** of the search tree (a constant amount of memory for each node can be assumed)

Computational complexity can therefore be evaluated as follows:

- worst-case **time** complexity: the highest number of nodes that are **generated** before a solution is found (if any)
- worst-case **space** complexity: the highest number of nodes that have to be **simultaneously** stored in memory

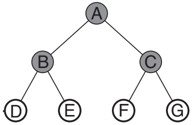
Breadth-first search: computational complexity

The computational complexity of BFS depends on two main factors:

- the number of successors of each node of the search tree
- the depth d of the **shallowest** solution (the one found by BFS)

Although different nodes can have a different number of successors (e.g., in 8-puzzle and route finding in maps), to simplify the evaluation of computational complexity a **constant** number b of successors, named **branching factor**, is considered.

An example for $b = 2$ (**binary tree**, up to depth 2):



Breadth-first search: computational complexity

For a **fixed** branching factor, computational complexity can be evaluated as a function of d only.

Time complexity: in the **worst case** the goal state is in the **last** node that is selected in step 2.2 to be expanded, among all the ones at depth d. This means that all the other nodes at depth d are expanded before.

The overall number of generated nodes can be computed by evaluating the number of nodes that are generated at each depth:

Depth	**Number of generated nodes**
0	1 (root node)
1	b
2	b^2
3	b^3
...	...
d	b^d
$d+1$	$b^{d+1} - b$
Total:	$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b)$

Breadth-first search: computational complexity

Space complexity: in the worst case **all** generated nodes remain in memory until a solution is found. It follows that space complexity equals time complexity.

The worst-case time and space complexity of BFS, for a search tree with branching factor b and shallowest solution at depth d, are therefore given by:

$$1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = \mathcal{O}(b^{d+1}) \ .$$

The computational complexity of BFS is thus **exponential** with respect to the depth of the shallowest solution.

In general, an exponential computational complexity denotes a very low efficiency.

Breadth-first search: computational complexity

An example: consider a search problem with the following features

- branching factor $b = 10$
- time for generating one node: 10^{-4} s
- storage required for a single node: 1,000 bytes

Worst-case time and space complexity of BFS, as a function of the depth d of the shallowest solution:

Depth	**Generated nodes**	**Time**	**Memory**
2	1,101	0.11 sec.	1 megabyte
4	111,101	11 sec.	106 megabytes
6	10^7	0.19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3, 523 years	1 exabyte

Properties of breadth-first search

Summary of BFS properties:

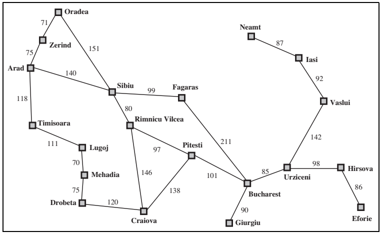
- **complete**: a solution is always found, if one exists
- **non-optimal**: it is not guaranteed that the solution with minimum path cost is found (if any), unless the path cost is a non-decreasing function of depth
- **exponential** time and space complexity, with respect to the depth of the shallowest solution

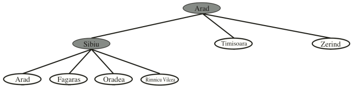
Depth-first search (DFS)

DFS expands first the **deepest** leaf node (in case there are several such nodes, a random choice is made).

This amounts to explore first one **whole** path; if no solution is found, another **whole** path is explored, and so on.

An example:

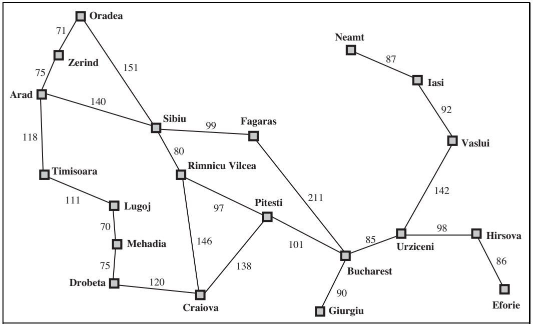




deepest leaf nodes: Arad, Fagaras, Oradea, Rimnicu Vilcea;
one of them is randomly chosen

Example of depth-first search (1/5)

Route finding on maps: getting from Arad to Bucharest.



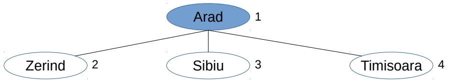
Example of depth-first search (2/5)

1. root node: the initial state, Arad; fringe = (Arad (1))
2.1 the fringe is not empty
2.2 the **deepest** node in the fringe (the root) is chosen (Arad)
2.3 Arad is not a goal state

Example of depth-first search (3/5)

2.4 Arad is removed from the fringe and expanded, generating three nodes having **identical** depth

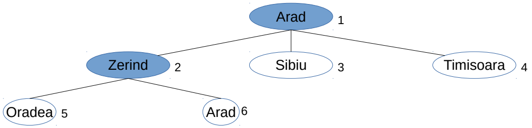
- the newly generated nodes must be inserted in the fringe (a queue) in the order in which they will be expanded by DFS; by definition, in DFS a newly expanded node has depth equal or higher than **all** the other nodes in the fringe; therefore, the newly expanded nodes are inserted **at the top** of the fringe
- since the newly generated nodes always have identical depth, they are inserted in the fringe in **any** order between themselves



Current fringe: (Zerind (2), Sibiu (3), Timisoara (4)).

Example of depth-first search (4/5)

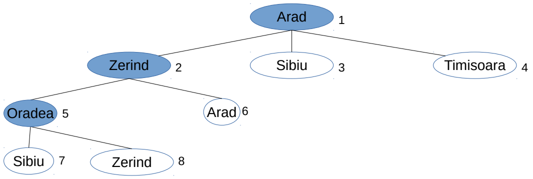
2.1 the fringe is not empty
2.2 the **first** node in the fringe is selected (it is guaranteed to be one of the deepest leaf nodes)
2.3 the corresponding state, Zerind, is not the desired state
2.4 Zerind is removed from the fringe and expanded, generating two nodes that are inserted at the top of the fringe



Current fringe: (Oradea (5), Arad (6), Sibiu (3), Timisoara (4)).

Example of depth-first search (5/5)

2.1 the fringe is not empty
2.2 the **first** node in the fringe is selected
2.3 the corresponding state, Oradea, is not the desired state
2.4 Oradea is removed from the fringe and expanded, generating two nodes that are inserted at the top of the fringe

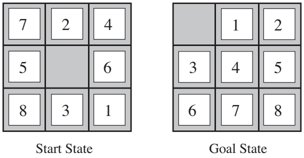


Current fringe:
(Sibiu (7), Zerind (8), Arad (6), Sibiu (3), Timisoara (4)).

And so on.

Exercise

Apply the DFS algorithm to the 8-puzzle problem, considering the initial and goal states below, and **expanding** the first **four** nodes of the search tree (i.e., execute the first four iterations of the **general tree-search algorithm**).



Some remarks on depth-first search

DFS can spend time along very long paths.

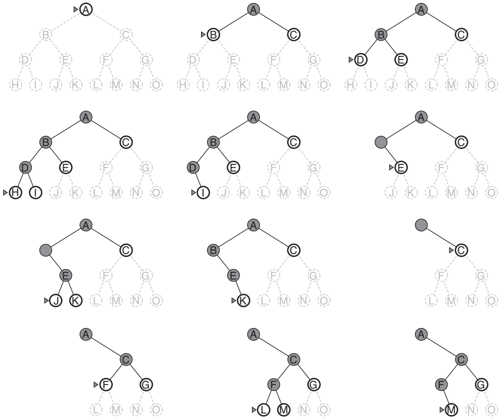
Infinite paths can also occur if **loops** are not avoided, e.g.: Arad $\rightarrow$ Zerind $\rightarrow$ Arad $\rightarrow$ Zerind ...

On the other hand, DFS has modest memory requirements

- if **all** the paths starting from a given node have been fully explored (avoiding loops) and no solution has been found, the sub-tree having such a node as the root can be **removed** from memory
- it follows that only a **single** path form the root to a leaf node needs to be stored in memory during the search, together with the unexpanded sibling nodes for each node on that path

Example of depth-first search

An example for a **binary** search tree, assuming each path has maximum depth 3 (avoiding loops), and that node `M` contains a goal state. Shaded nodes are the ones not yet generated. Ties (several nodes at the highest depth) are broken by choosing the left-most node.



Depth-first search: computational complexity

Worst-case computational complexity can be evaluated by assuming that

- all nodes have a constant branching factor b
- all solutions have the same depth m
- m is also the **maximum** depth of the search tree (avoiding loops)
- the goal state is in the **last** path that is explored

Time complexity: the above assumptions imply that **all** nodes up to depth m are generated before the solution is found.

Space complexity: only a **single** path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node in the path, needs be stored (see the example above).

Depth-first search: computational complexity

Under the above assumptions, the computational complexity of DFS as a function of the solution depth m is given by:

	Time complexity	Space complexity
Depth	**Generated nodes**	**Stored nodes**
0	1 (root node)	1 (root node)
1	b	b
2	b^2	b
...	...	...
m	b^m	b
Total:	$1 + b + \ldots + b^m = \mathcal{O}(b^m)$	$1 + mb = \mathcal{O}(m)$

Time complexity is therefore **exponential** with respect to the solution depth m, as that of BFS, but space complexity is **linear**.

Properties of depth-first search

Summary of DFS properties

- **complete**, if loops (infinite paths) are avoided
- **non-optimal**: a deeper, suboptimal solution could be found along a path that is explored before another one that may contain an optimal solution at smaller depth
- **exponential** time complexity, but **linear** space complexity, with respect to the solution depth

Other strategies

Uniform-cost: expands the leaf node with the lowest path cost

- **optimal**
- **complete**

Depth-limited: DFS with a depth limit D

- **complete**, provided that D is not smaller than the depth of the shallowest solution (which must be known in advance)
- allows finding shallower solutions first, reducing time complexity with respect to DFS
- **not optimal**

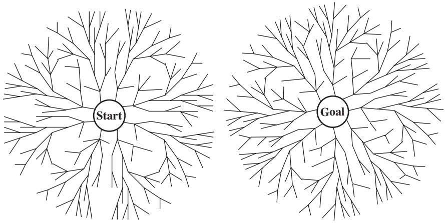
Iterative-deepening depth-first: repeated depth-limited search with increasing values of D, until a solution is found

- **not optimal**
- **complete**

Other strategies

Bidirectional: simultaneously searching **forward** from the initial state and **backwards** from the goal state, with alternate node expansions, until the two searches meet, i.e., a common state is found in their frontiers

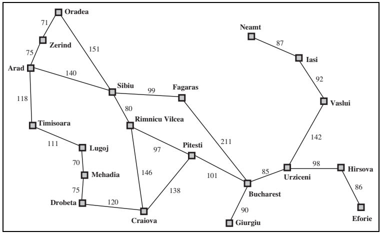
- requires **reversible** actions (e.g., 8-puzzle, and route finding in maps with no one-way roads)
- DFS is not suitable, since at each step it may explore different paths in the two searches



Avoiding repeated states

Search algorithms may waste time by expanding **different** nodes associated with the **same** state. This may happen, e.g., when:

- actions are **reversible**, which allows **loops** like: Arad → Zerind → Arad → Zerind . . .
- **cyclical paths** exist (loops are a particular case), e.g.: Arad → Zerind → Oradea → Sibiu → Arad
- **different** paths can lead to the **same** state, e.g.: Arad → Sibiu, and Arad → Zerind → Oradea → Sibiu



Avoiding repeated states

Four solutions of increasing complexity can be adopted.
When a node `n` is being expanded:

1. if reversible actions exist, discard the newly generated node containing the same state of the **parent** node of `n` (this avoids loops)
2. discard all children nodes containing states already present **in the same path** from the root to `n` (this avoids cyclical paths)
3. discard all children nodes containing states already present **in the current search tree** (not suitable to DFS, which does not store all generates nodes)
4. discard all children nodes containing states **previously** generated, even if not present in the current search tree

Avoiding repeated states

The above solutions require to compare every newly generated node with some other nodes, which can affect computational complexity:

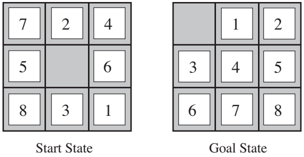
- solution 1 requires a single comparison
- solution 2 requires a number of comparisons equal to the depth of the parent node `n`
- solution 3 requires a comparison with **all** the nodes in the **current** search tree: its **time complexity** is **exponential** for strategies with exponential space complexity (like BFS)
- solution 4 requires to store the states of **all previously generated** nodes (increasing the space complexity of DFS)

Using strategies 3 and 4 it is better to remove the node with the **highest** path cost.

Effectiveness of uninformed search: an example

One may think that the high computational complexity of uninformed search strategies is an issue only for real-world problems, not for toy ones.

Consider again 8-puzzle, apparently a very simple toy problem:



How long does it take to solve it using, e.g., BFS?

Effectiveness of uninformed search: an example

Some facts about 8-puzzle:

- the state space contains $9! = 362,880$ distinct states (only $9!/2 = 181,440$ are reachable from any given initial state)
- the average solution depth (over all possible pairs of initial and goal states) is about 22
- the average branching factor b (over all possible states) is about 3 (from each state 2 to 4 actions can be performed)

How many nodes does BFS generate and store, when the shallowest solution has depth $d = 22$ (i.e., in the **average** case)?

Remember that the worst-case time and space complexity of BFS is $\mathcal{O}(b^{d+1})$, which for $d = 22$ amounts to $3^{23} \approx 9 \times 10^{10}$...

For instance, taking into account that representing a state requires at least $\lceil \log_2 9! \rceil = 19$ bits, storing 3^{23} states requires about $19 \times 9 \times 10^{10}$ bits, i.e., more than 200 GB...

Exercise

1. Implement the **general tree-search algorithm**, and the related data structures, in a programming language of your choice
2. Implement the **additional**, specific functions for breadth-first, depth-first and uniform-cost search
3. Implement the **additional**, specific data structures and functions for the 8-puzzle problem, and the route finding problem in the Romania map
4. Run the above search algorithms on specific problem instances, and evaluate the number of generated nodes and the maximum number of nodes simultaneously stored in memory

Informed search

Uninformed search systematically explores the search space (e.g., expanding the shallowest or the deepest node first), without relying on information (if any) about what nodes are more “promising” than others towards the solution.

When such kind of information is available, it can be exploited to improve search effectiveness and efficiency.

This approach is named **best-first search**.

Best-first search

Best-first search is based on quantitatively evaluating how "promising" a given node `n` is towards a solution, through a suitable **node evaluation function** $f(\mathtt{n})$ (conventionally, lower values of f correspond to "better" nodes).

Different definitions of $f(\mathtt{n})$ lead to different, specific best-first search strategies, for instance:

- greedy search
- A*-search and its many variants (iterative-deepening A*, memory-bounded A*, etc.)

Best-first search

Once a suitable f(`n`) (i.e., a specific best-first search strategy) has been defined, the corresponding search algorithm can be implemented using **the same general tree-search algorithm** presented above.

Best-first search can be easily implemented by sorting nodes in the fringe for increasing values of f(`n`): this allows the node `n` with the lowest f(`n`) (the "best" node) to be selected for expansion at each iteration.

Best-first search

To define $f(\mathtt{n})$, a very useful information is the cost of the actions that will lead from any given node `n` to a goal state.

Although in non-trivial problems the **exact** cost is usually unknown, often an **estimate** of it can be easily computed.

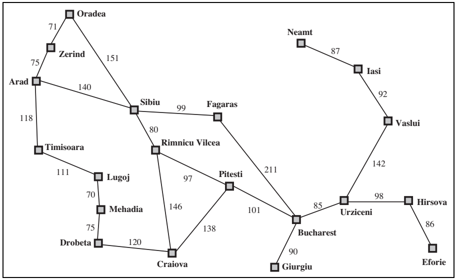
The estimated cost is formalized as a function $h(\mathtt{n})$. By definition, $h(\mathtt{n}) = 0$ if `n` contains a goal state (this is the only case in which the cost is **exactly** known).

For historical reasons $h(\mathtt{n})$ is named **heuristic function**, and search strategies based on it are also named **heuristic search**.

Heuristic search is one of the early achievements of AI (dating back to the 1950's), but is still widely used in real-world problems and investigated by researchers in AI.

Heuristic functions: an example

Consider the problem of route finding in maps, e.g., finding the **shortest** route form Arad to Bucharest using the information on the map below:



Since the goal is to find the shortest route, the cost of the actions is evaluated as the route length (e.g., in Km). Defining an heuristic function for this problem amounts therefore to estimating the **distance** between any given city and the destination.

Heuristic functions: an example

An **easy-to-compute** estimate for this kind of problem is the **straight-line distance** (e.g., it can be easily computed from the geographical coordinates of each city).

If the destination (goal state) is Bucharest, the heuristic function $h(\mathtt{n})$ is defined as the straight-line distance from the city (state) associated with node `n` to Bucharest:

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

Greedy best-first search

The simplest best-first search strategy: expanding the node that **appears** to be the **closest** to the solution.

Since usually the exact cost is unknown, this strategy is implemented using the estimated cost, i.e., the heuristic function $h(\mathtt{n})$.

Accordingly, the node evaluation function is simply defined as:

$$f(\mathtt{n}) = h(\mathtt{n})$$

This strategy is called “greedy” since it favours the partial solution that **appears** to be the closest to the goal state (since $h(\mathtt{n})$ is only an estimate), but, as it will become more clear later, this is not an optimal choice.

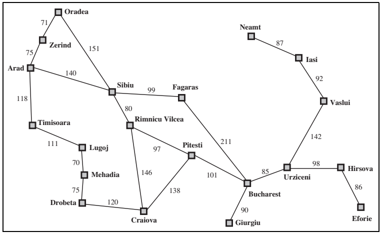
Greedy best-first search: an example

Consider again the problem of finding the shortest route from Arad to Bucharest, using the straight-line distance heuristic.

In the following, the search tree built by greedy search, until a solution is found, is shown, including the value of $f(\texttt{n})$ for each node. The node selected for expansion is highlighted by an arrow.

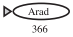
Remember that using the **general tree-search algorithm** a solution is found when a node containing a goal state is selected to be **expanded**, **not** when it is **generated** by the expansion of its parent node.

Greedy best-first search: an example

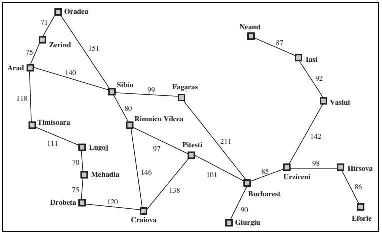


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(a) The initial state

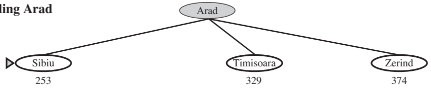


Greedy best-first search: an example

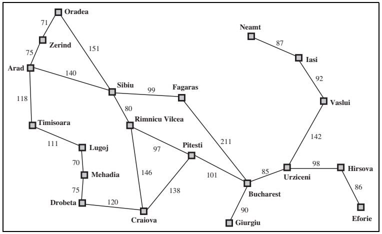


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(b) After expanding Arad

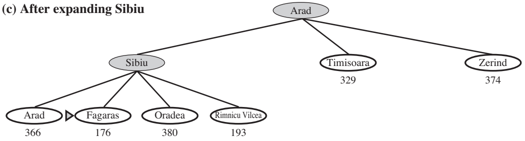


Greedy best-first search: an example

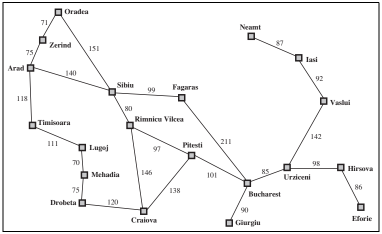


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(c) After expanding Sibiu

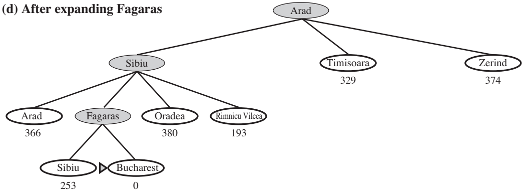


Greedy best-first search: an example



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

(d) After expanding Fagaras



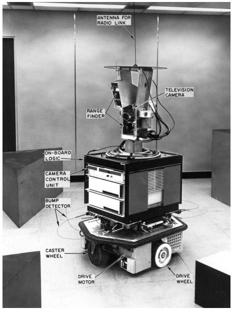
Properties of greedy best-first search

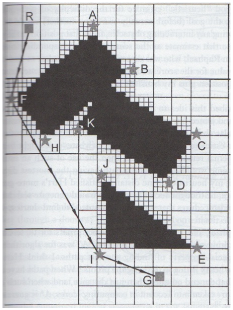
Summary of greedy search properties

- **complete**, if cyclical paths (including loops) are avoided
- **non-optimal**, e.g., in the above example the route found by greedy search is not the shortest one (Arad $\rightarrow$ Sibiu $\rightarrow$ Rimnicu Vilcea)
- worst-case time and space complexity are **exponential** in the depth of the shallowest solution m, for a constant branching factor b: $\mathcal{O}(b^m)$

A* search

A* is the most relevant best-first search strategy. It was devised in the 1960s for robot navigation tasks.





Many variants of A* have been proposed since then to achieve different trade-offs between effectiveness and efficiency.

A* search

Greedy search chooses for expansion the node `n` which appears closest to the goal state, i.e., such that the estimated cost of the actions from `n` to a goal state is minimum. However, it disregards the cost of the actions from the root to `n`.

A* uses instead an estimate of the **total** cost of the action sequence from the root to a goal state through `n`, defined as the sum of the path cost of `n` (which is **exactly** known) and the estimated cost from `n` to the solution.

The corresponding node evaluation function is defined as:

$$f(\mathtt{n}) = \underbrace{g(\mathtt{n})}_{\text{path cost from root to } \mathtt{n}} + \underbrace{h(\mathtt{n})}_{\text{estimated cost from } \mathtt{n} \text{ to the solution}}$$

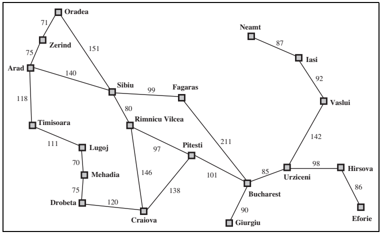
A* search: an example

In the following the search tree built by A* is shown for the same problem of the previous example (Arad-Bucharest, using the straight-line distance heuristic). The value of $f(\mathtt{n}) = g(\mathtt{n}) + h(\mathtt{n})$ is also shown for each node.

Note that after the fourth iteration (the expansion of the node containing the state Fagaras) a leaf node containing the goal state Bucharest is generated. However, it is not selected for expansion at the next iteration (and thus a solution is not found yet), since it is not the node with the minimum value of $f(\mathtt{n})$.

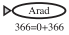
During the expansion of the node containing the state Pitesti in the fifth iteration, a **different** node containing the goal state Bucharest is generated. This latter node is selected for expansion at the next iteration, since it has the minimum value of $f(\mathtt{n})$, and, since it contains a goal state, a solution is found and A* terminates.

A* search: an example

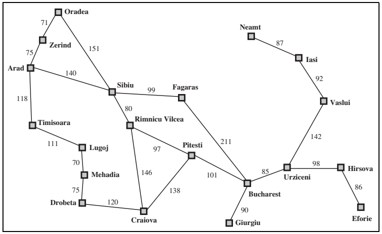


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(a) The initial state

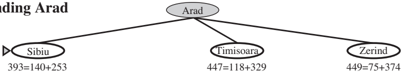


A* search: an example

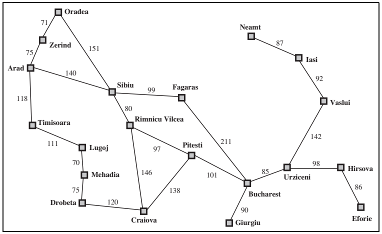


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(b) After expanding Arad

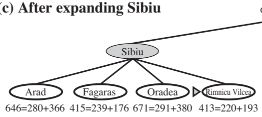


A* search: an example

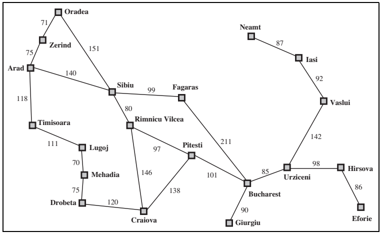


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(c) After expanding Sibiu

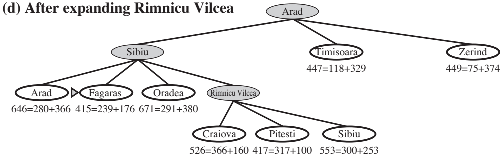


A* search: an example

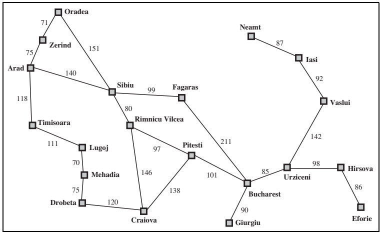


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(d) After expanding Rimnicu Vilcea

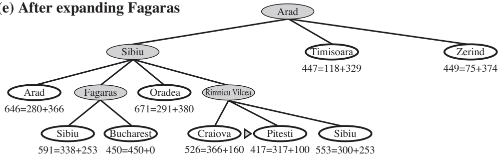


A* search: an example

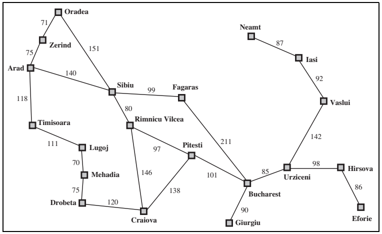


Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

(e) After expanding Fagaras

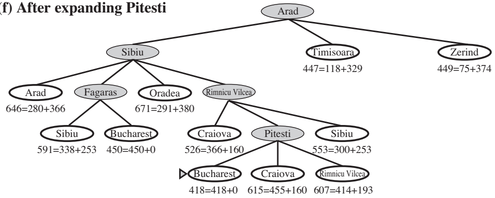


A* search: an example



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

(f) After expanding Pitesti



Properties of A* search

Summary of A* properties

- **optimal**, provided that the heuristic is **admissible**, i.e., it never overestimates the exact cost to the solution (e.g., the straight-line distance is admissible for route finding in maps)
- **complete**, as well as **optimally efficient** (i.e., it expands the **lowest** number of nodes) for any admissible heuristic, among algorithms that extend search paths from the root
- worst-case time and space complexity are **exponential** in the depth of the shallowest solution m, for a fixed branching factor b: $\mathcal{O}(b^m)$; nevertheless, in practice A* is often **much more efficient** (i.e., it generates a much smaller number of nodes) than other uninformed and informed search strategies

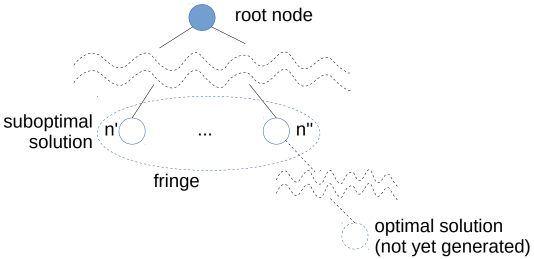
Proof of A* optimality

Assume the fringe contains one leaf node $\mathtt{n}'$ with a suboptimal goal state, and no leaf node with an optimal goal state.

Can A* ever select $\mathtt{n}'$ to be expanded, thus returning it as a **suboptimal** solution?

First, note that some leaf node $\mathtt{n}'' \neq \mathtt{n}$ in the path toward an optimal solution **must** exist in the fringe.

We have to consider therefore the following scenario:



Proof of A* optimality

Denoting with C^* the cost of an optimal solution, the above assumptions imply:

1. $h(\mathtt{n}') = 0$ ($\mathtt{n}'$ contains a goal state)
2. $f(\mathtt{n}') = g(\mathtt{n}') + h(\mathtt{n}') = g(\mathtt{n}') > C^*$
   ($\mathtt{n}'$ contains a sub-optimal goal state)
3. $f(\mathtt{n}'') = g(\mathtt{n}'') + h(\mathtt{n}'') \leq C^*$ ($\mathtt{n}''$ is in the path toward an optimal solution, and h is admissible, thus $f(\mathtt{n}'')$ does not overestimate the cost of any solution reachable through $\mathtt{n}''$)

In turn, expressions **??** and **??** above imply:

$$f(\mathtt{n}') > C^* \geq f(\mathtt{n}'')$$

This means that $\mathtt{n}'$ **cannot** be selected for expansion, and thus A* **cannot** return a suboptimal solution.

Improving A* search

- "Good" heuristics (discussed later) can reduce time and memory requirements, especially with respect to uninformed search
- In many practical problems even A* is infeasible: memory requirements are the main drawback. Several variants have been devised:
  - using **non-optimal** A* variants that find suboptimal solutions quickly
  - using A* variants with reduced memory requirements and a small increase in execution time, but still optimal

Defining heuristic functions

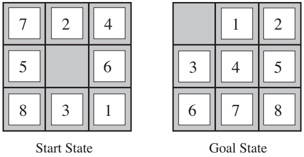
Intuitively, the more accurate is the estimate of the cost to the solution from a given node provided by the heuristic function, the more efficient a best-first algorithm is.

Defining a **good** (i.e., accurate) heuristic is therefore crucial for informed search. Moreover, heuristics have to be **admissible** to guarantee the optimality of A*.

Defining heuristic functions: examples

We have seen that a possible heuristic for route finding in maps is the straight-line distance.

Consider now the 8-puzzle problem. Remember that about 9×10^{10} nodes are generated on average by breadth-first (uninformed) search when the solution is at the average depth of 22: therefore a good heuristic can be of great practical help also in this toy problem.



As an **exercise**, try to devise admissible heuristic functions for the 8-puzzle problem.

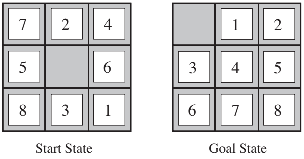
Defining heuristic functions

Well-known admissible heuristics for k-puzzle are the following:

- number of misplaced tiles (in the following, $h_1(\mathtt{n})$)
- **city block** or **Manhattan** distance: sum of the "distances" of each tile from its goal position ($h_2(\mathtt{n})$)

An example: values of h_1 and h_2 for the start state below on the left, with respect to the goal state on the right

- $h_1(\text{start state})$: 8 (all 8 tiles are misplaced)
- $h_2(\text{start state})$: $3+1+2+2+2+3+3+2=18$ (tiles 1 to 8)



Defining or choosing heuristic functions

For some problems it may be not straightforward to define a heuristic function. In that case a general approach is to set $h(\mathtt{n})$ equal to the **exact** cost of a **relaxed** version of the problem at hand, which may be easy to compute.

Some examples:

- k-**puzzle**: by relaxing the constraint that tiles can move only to a **free** adjacent square, and allowing them to move to **any adjacent** square, one obtains $h_2(\mathtt{n})$ (see above)
- k-**puzzle**: allowing tiles to move to **any** square (even non-adjacent and occupied ones), one obtains $h_1(\mathtt{n})$
- **route finding in maps**: allowing one to directly reach any city (not only adjacent ones), and to move straight to them (not only along roads), one obtains the straight-line distance heuristic

Defining or choosing heuristic functions

On the other hand, for some problems it can be possible to define several admissible heuristics $h_1, \ldots, h_p$ (e.g., h_1 and h_2 for 8-puzzle).

In this case one can choose or define a single heuristic h which **dominates** all the other ones, i.e.:

$$\textbf{for each} \text{ node } \mathtt{n},\ h(\mathtt{n}) \geq h_i(\mathtt{n}), i = 1, \ldots, p.$$

It is easy to see that such a heuristic is admissible, and provides a more accurate estimate of the cost to the solution than $h_1, \ldots, h_p$.

To this aim, h can be defined as follows:

- if there is a dominating heuristic among $h_1, \ldots, h_p$, choose it as the heuristic for the problem at hand
- otherwise, for a given node `n` use the following heuristic:

  $$h(\mathtt{n}) = \max\{h_1(\mathtt{n}), \ldots, h_p(\mathtt{n})\},$$

  which dominates by definition $h_1, \ldots, h_p$

Evaluating heuristic functions

To evaluate the quality of heuristic functions the concept of **effective branching factor** (denoted as b^*) is used:

- let N be the number of nodes generated by A* for a given problem, and d be the depth of the (optimal) solution
- b^* is defined as the branching factor of a **uniform** tree of depth d containing N nodes, which is the solution of the equation:

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

The **lower** the value of b^*, the better the heuristic.

Since b^* depends on the problem **instance**, it is usually evaluated **empirically** as the **average** over a set of instances.

Evaluating heuristic functions

As an example, the table below reports the results of an empirical evaluation of the effective branching factor of heuristics h_1 and h_2 for the 8-puzzle (used in A*), and, for comparison, of one of the most efficient uninformed search strategies, iterative-deepening depth-first search (IDS).

The comparison is made on 600 randomly generated problem instances with solution depth $d = 4, 8, \ldots, 24$ (100 instances for each depth value). The symbol – means that IDS could not terminate due to memory overflow. It is clear that h_2 is significantly better than h_1, and that uninformed search (IDS) is unfeasible even for 8-puzzle.

	search cost (expanded nodes)			effective branching factor		
d	IDS	A* (h_1)	A* (h_2)	IDS	A* (h_1)	A* (h_2)
4	112	13	12	2.87	1.48	1.45
8	6,384	39	25	2.80	1.33	1.24
12	3,644,035	227	73	2.78	1.42	1.24
16	–	1,301	211	–	1.45	1.25
20	–	7,276	676	–	1.47	1.27
24	–	39,135	1,641	–	1.48	1.26