★ (1) internet introduction

(2) general summary by Sveva Pepe
with a lot of exercises and other material
by me

(3) notes by Niccolò Brandizzi

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the search algorithms in AI.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

- **Search:** Searchingis a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
    a. **Search Space:** Search space represents a set of possible solutions, which a system may have.
    b. **Start State:** It is a state from where agent begins **the search**.
    c. **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

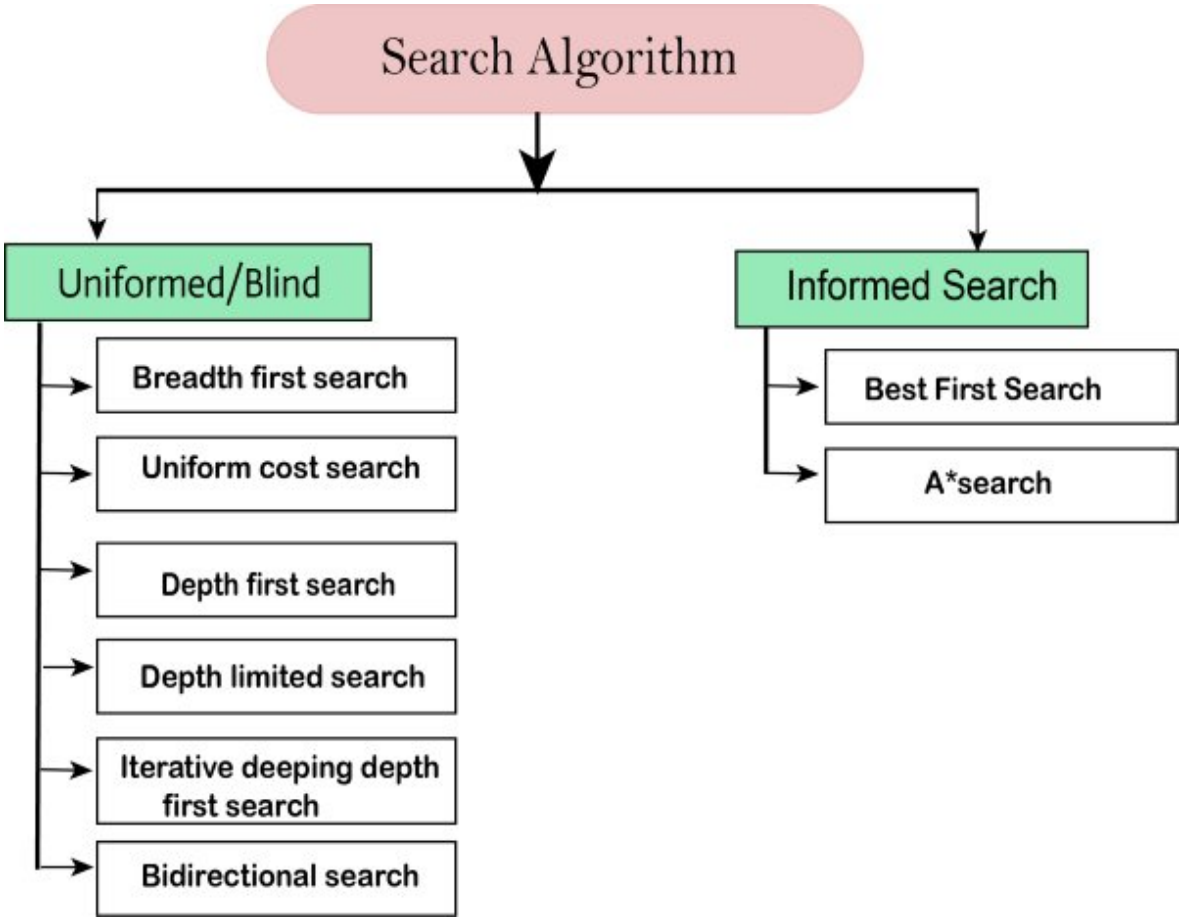
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search.It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve much complex problem which could not be solved in another way.

An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

Uninformed algorithms

Uninformed strategies use only the information available in the problem definition:

- Breadth-first search
- Depth-first search
- Depth limited search
- Iterative deepening search

stack → LIFO

input order: A, B, C
output order: C, B, A

queue → FIFO ← A B C ←

input order: A, B, C
output order: A, B, C

*[**b** = maximum branching factor of the search tree,*

***d** = depth of the least-cost solution,*

***m** = maximum depth of the state-space,*

***l** = limited depth]*

Breadth-first search

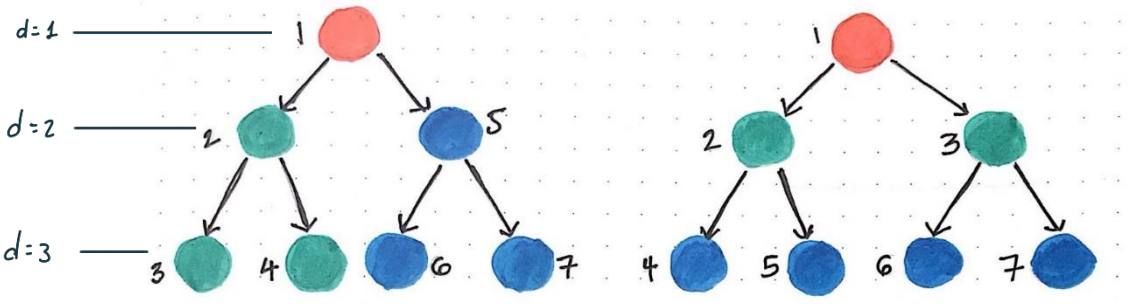
The tree is explored looking first at the root-node, then at all children of the root node, then at their successors and so on and so forth

Due to explore nodes, the iterative implementation uses a FIFO strategy.

It is complete; it is optimal, that means it finds a solution with the minimum amount of actions, finding the shallowest solution; it finds an optimal solution in $O(b^d)$, and it occupies $O(b^d)$ of memory.

when it scans the entire tree

It is optimal since you explore all the nodes and doing so you will not miss an eventual shallower solution



Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).

LIFO
stack

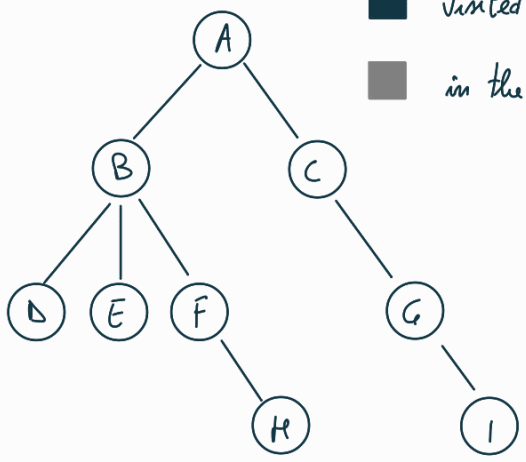
Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

FIFO
queue

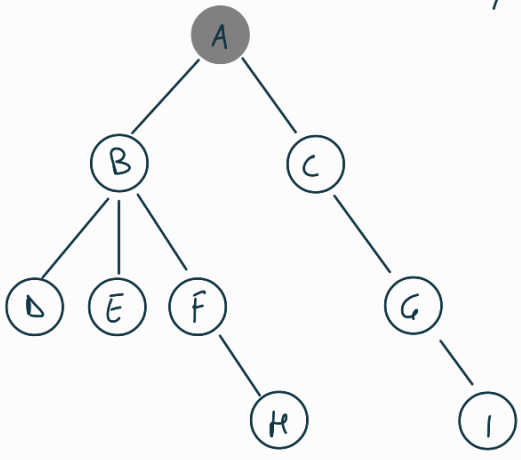
Breadth-first search example

① visited / in the queue



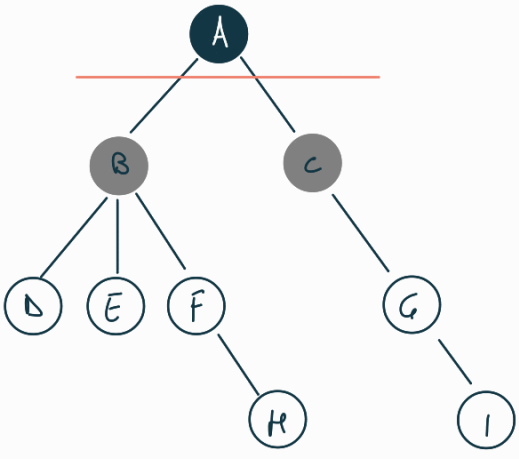
[A] push root node in the queue;
only one node in the queue

②

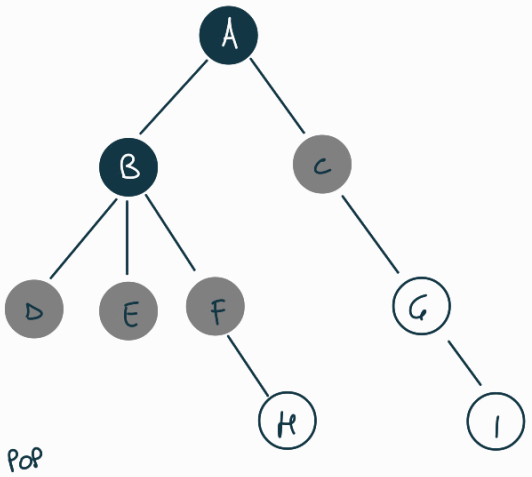


[ ] pop the root node;
mark it as visited;
[B, C] push its children

③



④

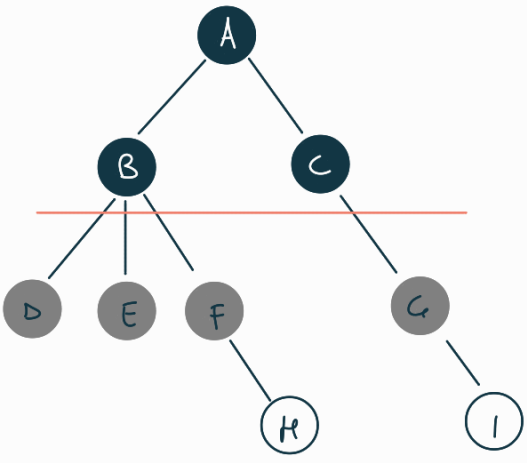


pop
[B, C]

[C, D, E, F]
push

pop
[queue]
push

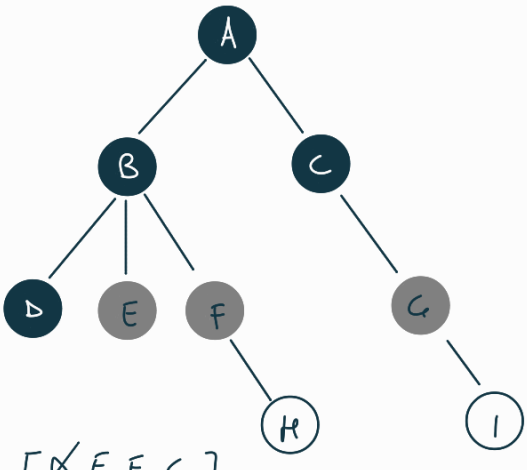
⑤



[C, D, E, F]

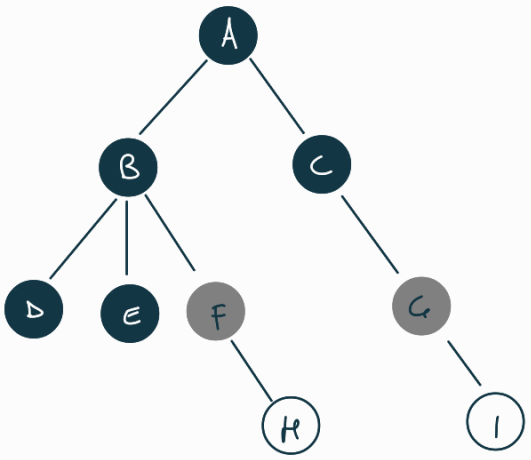
[D, E, F, G]

⑥



[E, F, G]

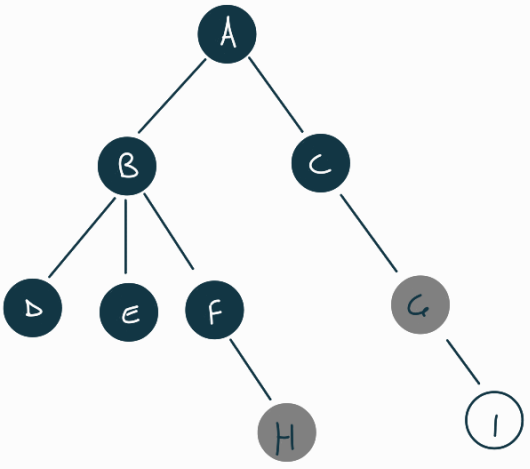
7



[~~E~~, F, G]

[F, G]

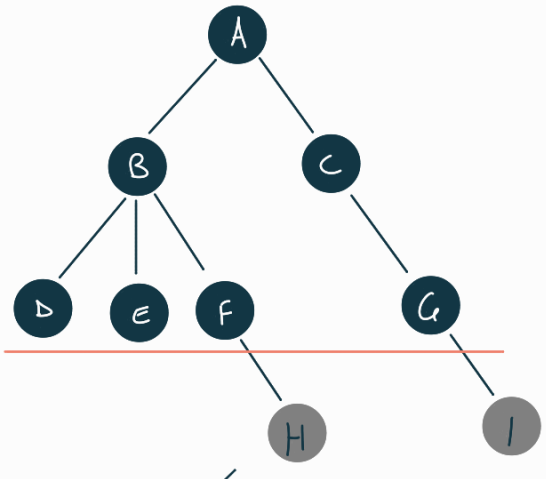
8



[~~F~~, G]

[G, H]

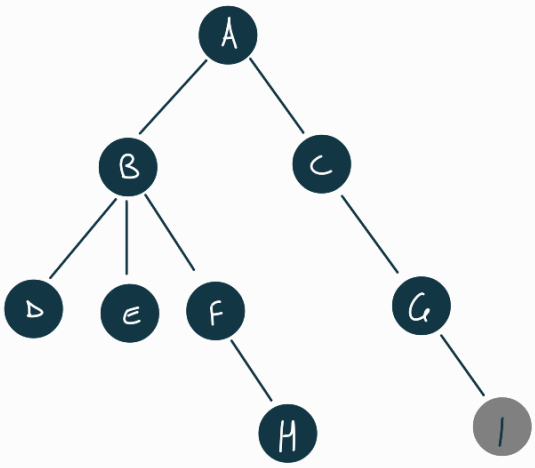
9



[~~G~~, H]

[H, I]

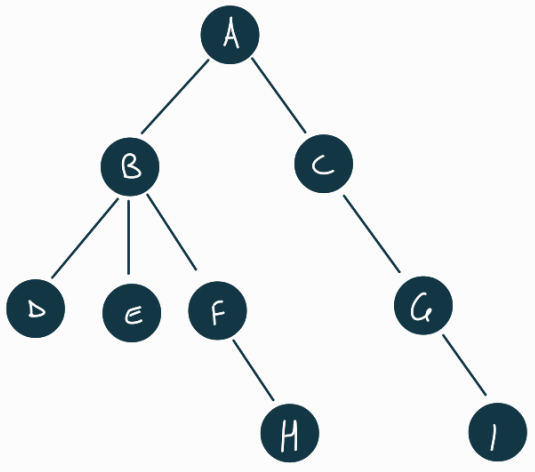
10



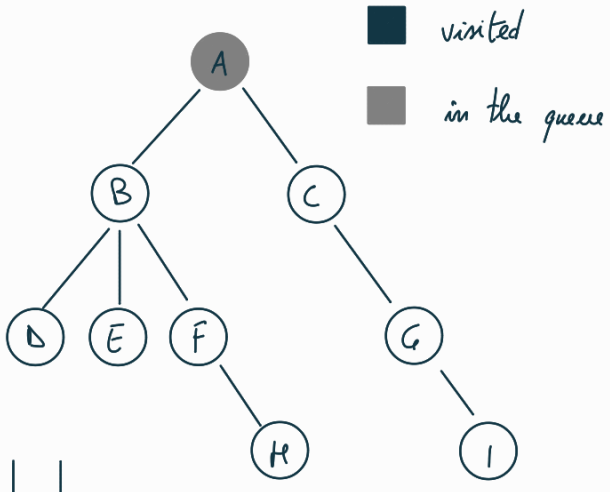
[~~H~~, I]

[I]

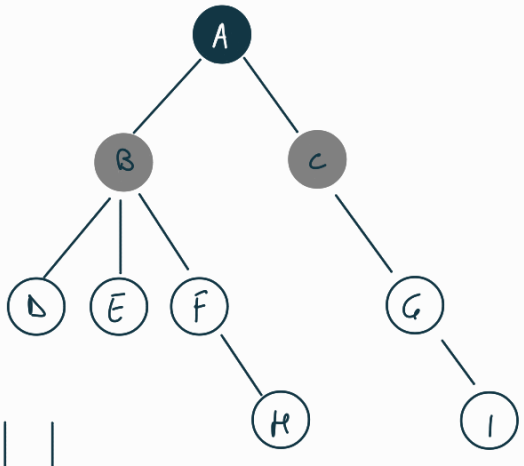
11



Depth-first search example



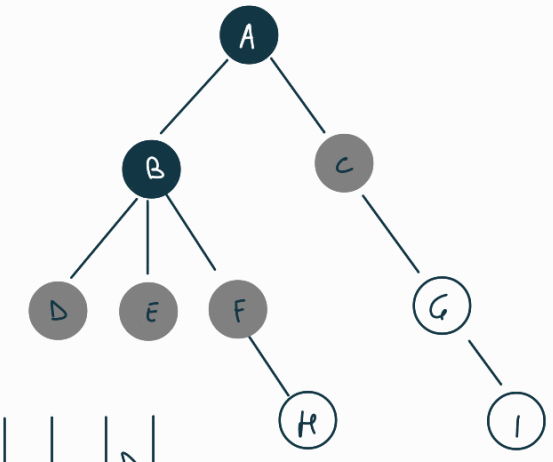
push root node in the stack
only one node in the stack

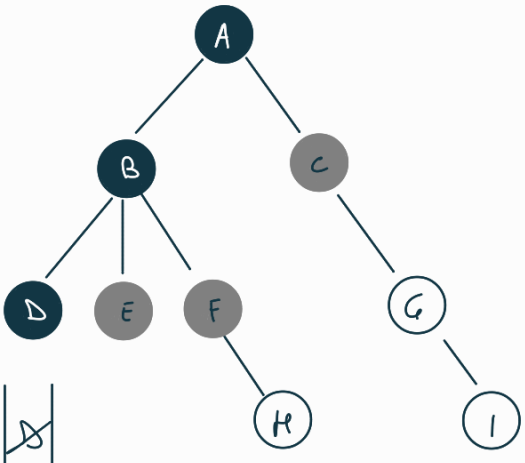


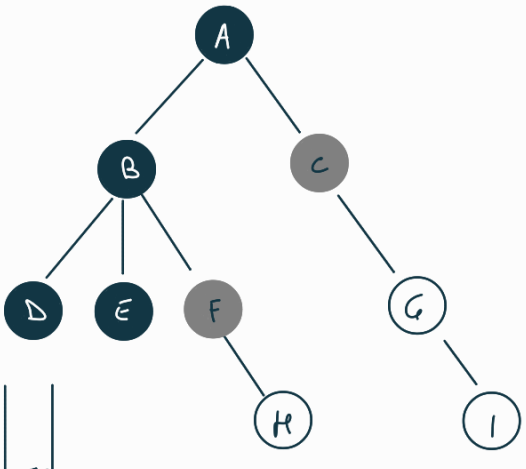
pop the root node;
mark it as visited;
push its children (the push order will change the search direction: first left, then right → pop right first, then left)

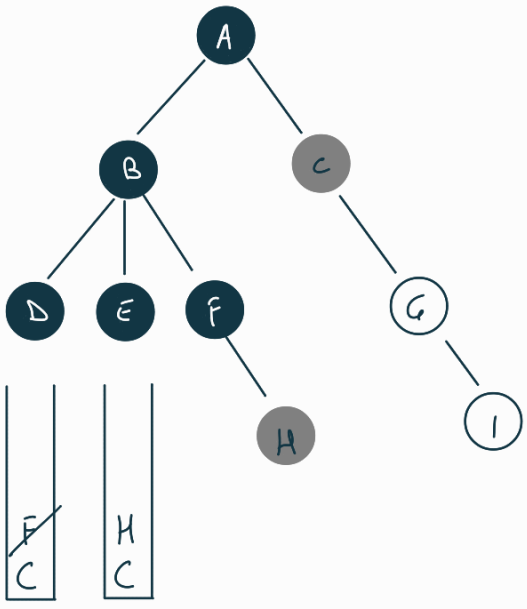
stack
LIFO

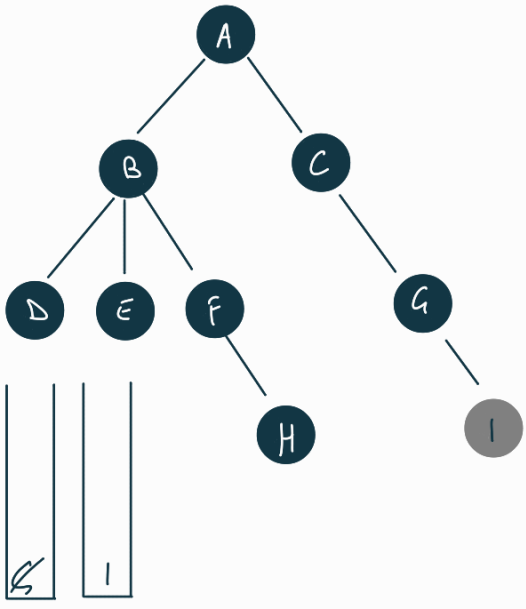
last in = B
first out = B

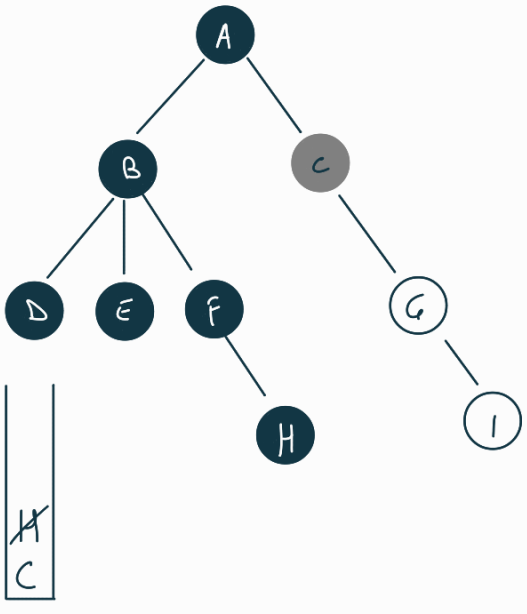


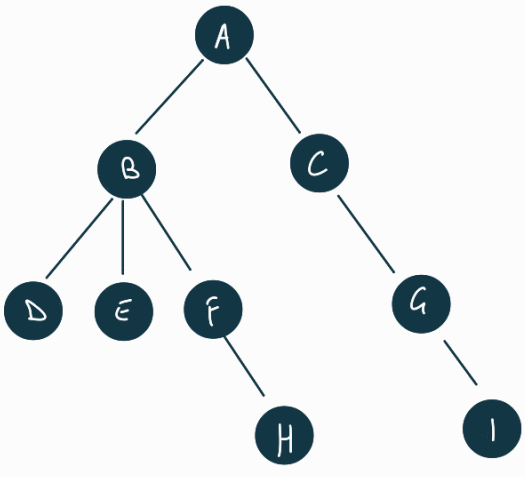


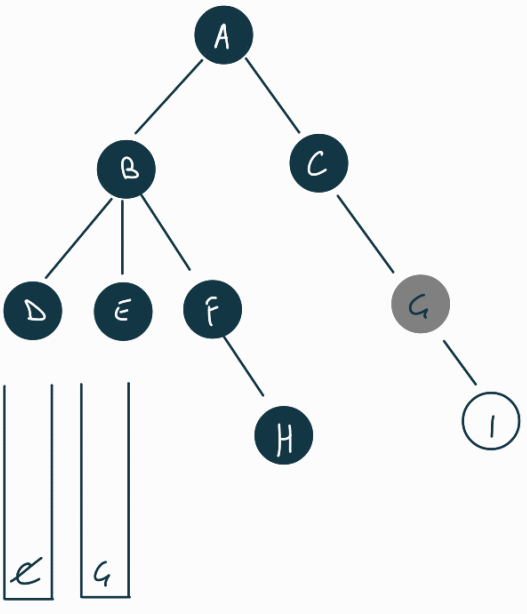



A
B
C
D
E
F
G
H
I
F
C
H
C


A
B
C
D
E
F
G
H
I
G
I


A
B
C
D
E
F
G
H
I
H
C


A
B
C
D
E
F
G
H
I


A
B
C
D
E
F
G
H
I
G

Uniform-cost: expands the node with the lowest path cost ($g(n)$).

Uniform-cost is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination.
Nodes are expanded, starting from the root, according to the minimum cumulative cost $g(n)$. The uniform-cost search is then implemented using a priority queue with $p(n) = g(n)$

Time and memory complexities are $O(b^{C^*/\varepsilon})$ where:

C^* is the cost of the optimal solution

ε is the least cost of a step

Pseudocode:

```
insert the root node into the priority queue
while the queue is not empty:
    n ← remove the element with the highest priority
    if the removed node n is the destination:
        print total cost and stop the algorithm
    else
        for each child c in children(n):
            priority(c) = priority(n) + cost(n, c)
            enqueue c
```

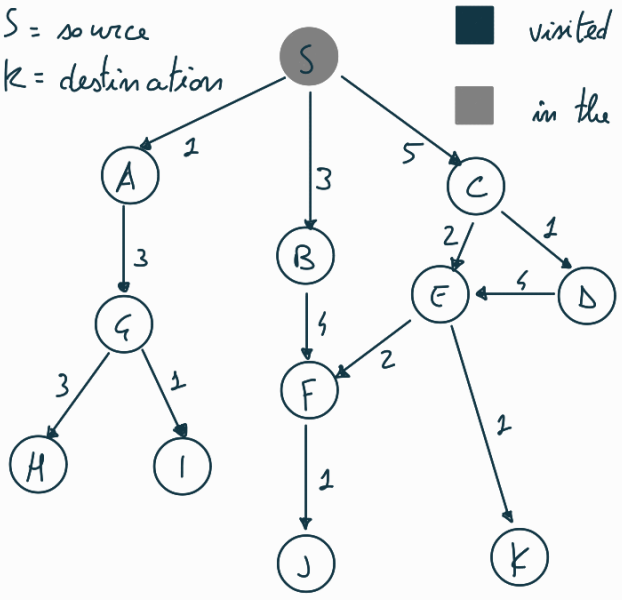
enqueue all the children of n to the priority queue with their cumulative cost from the root as priority

Uniform-cost search example

S = source
K = destination

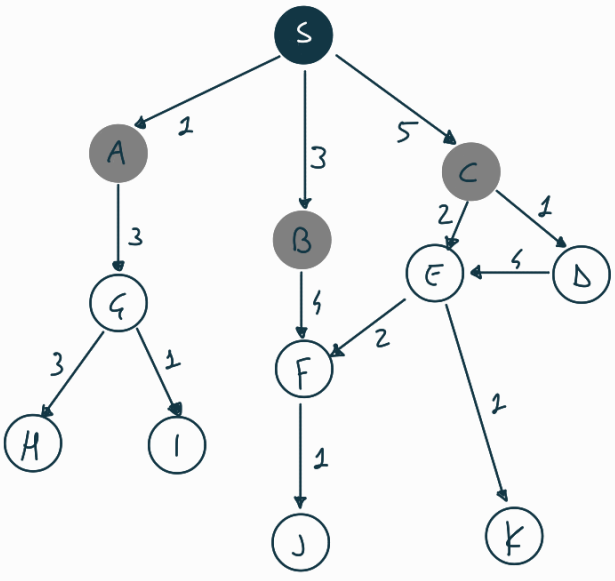
visited

in the queue

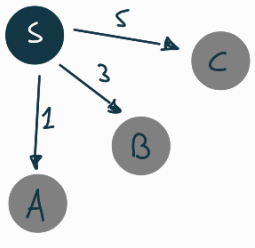


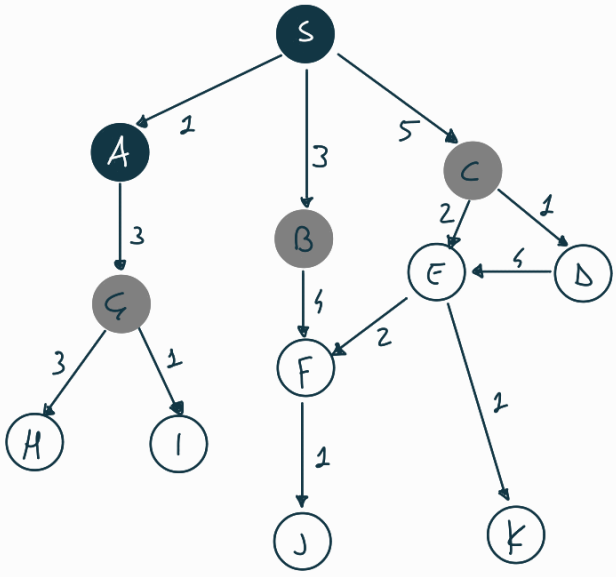
S(p=0)

pop S and mark it as visited
add its children to the queue



A(p=1); B(p=3); C(p=5)





POP

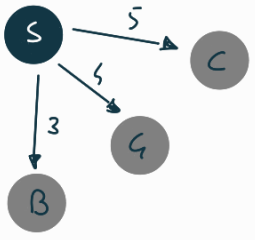
~~A(p=1)~~; B(p=3); C(p=5)

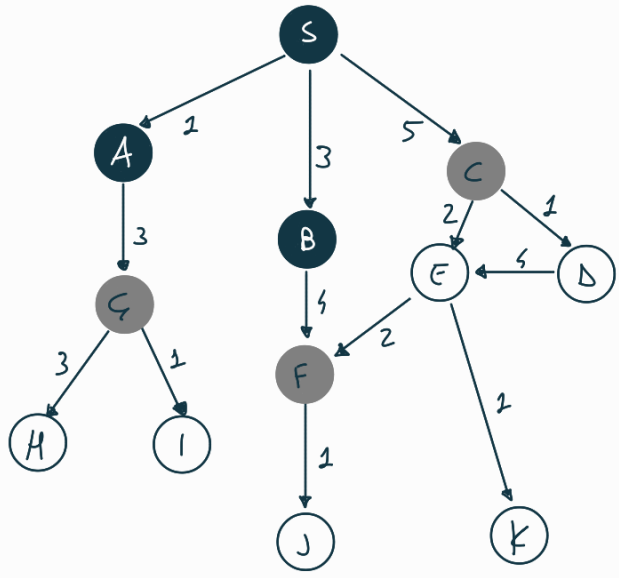
PUSH

B(p=3); C(p=5); G(p=1+3=4)

ORDER BASED ON THE PRIORITY

B(p=3); G(p=4); C(p=5)





POP

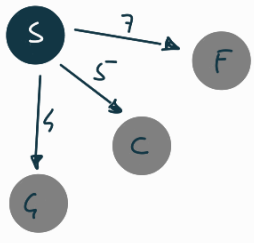
~~B(p=3)~~; G(p=4); C(p=5)

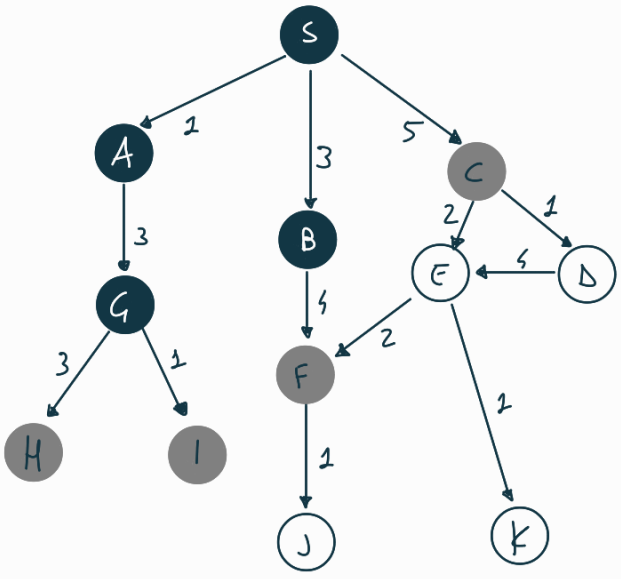
PUSH

G(p=4); F(p=3+4=7); C(p=5)

ORDER BASED ON THE PRIORITY

G(p=4); C(p=5); F(p=7)





POP

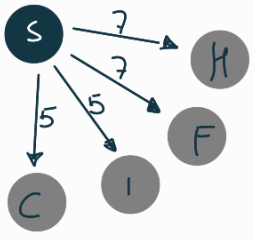
~~C(ρ=5)~~; C(ρ=5); F(ρ=7)

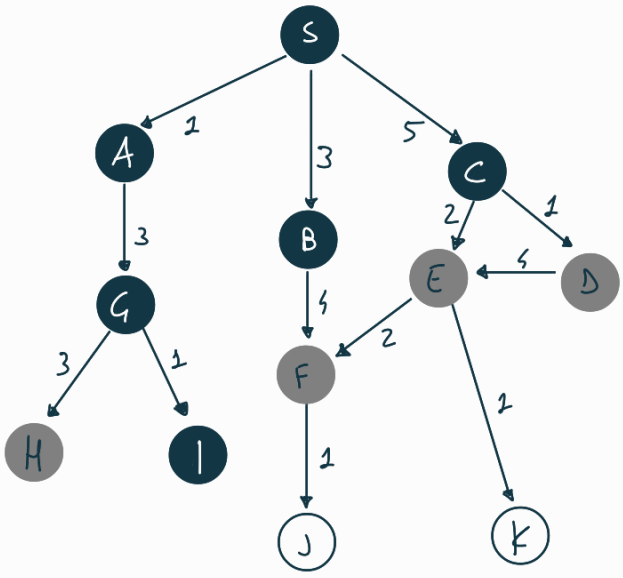
PUSH

C(5); F(7); H(4+3); I(4+1)

ORDER BASED ON THE PRIORITY

C(5); I(5); F(7); H(7)





POP

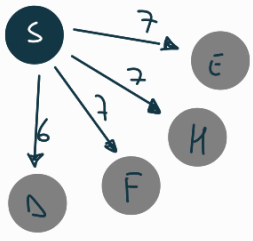
~~I(5)~~; D(6); F(7); H(7); E(7)

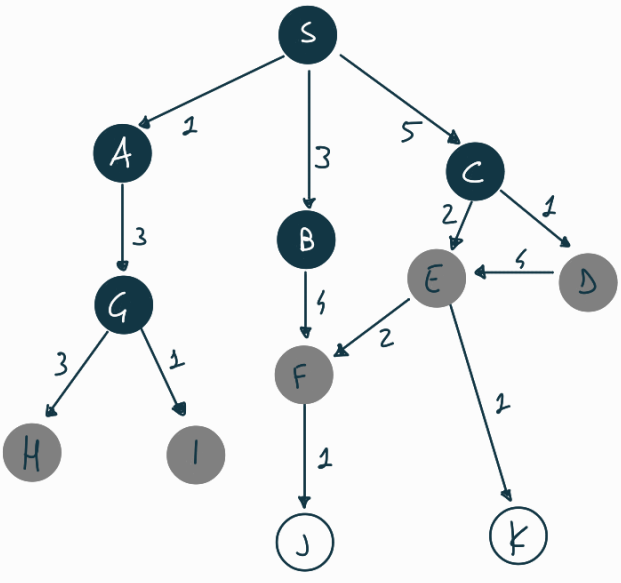
PUSH

D(6); F(7); H(7); E(7)

ORDER BASED ON THE PRIORITY

D(6); F(7); H(7); E(7)





POP

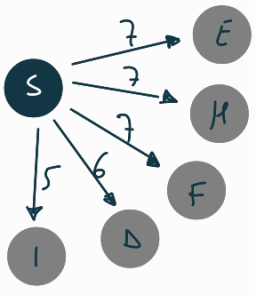
~~C(5)~~; I(5); F(7); H(7)

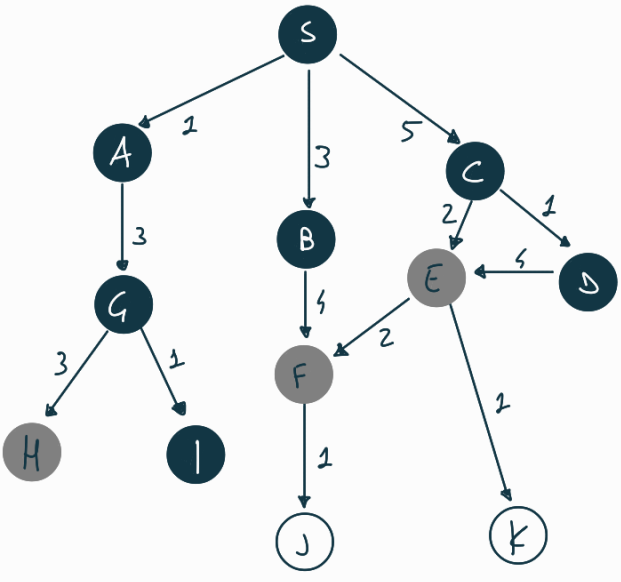
PUSH

I(5); F(7); H(7); E(5+2); D(5+1)

ORDER BASED ON THE PRIORITY

I(5); D(6); F(7); H(7); E(7)





POP

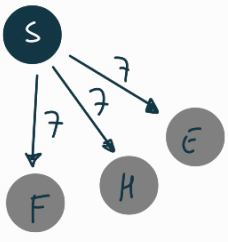
~~D(6)~~; F(7); H(7); E(7)

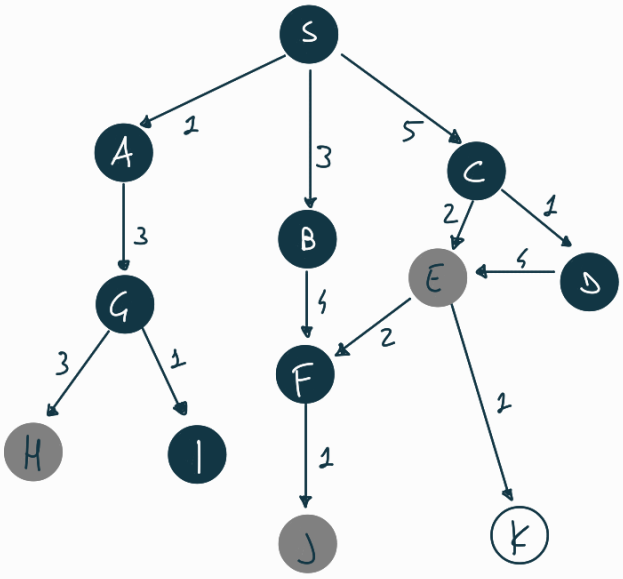
PUSH

F(7); H(7); E(7)

ORDER BASED ON THE PRIORITY

F(7); H(7); E(7)





POP

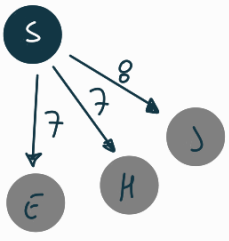
~~E(7)~~; H(7); E(7)

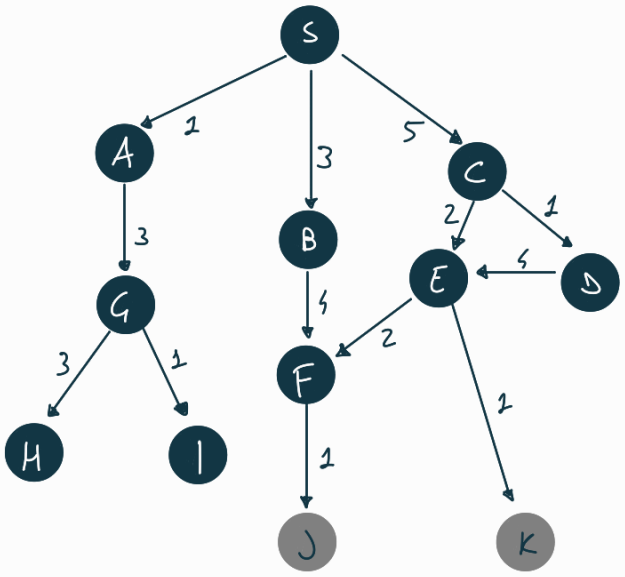
PUSH

H(7); E(7); J(7+1)

ORDER BASED ON THE PRIORITY

H(7); E(7); J(8)





POP

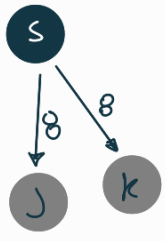
~~E(7)~~; J(8)

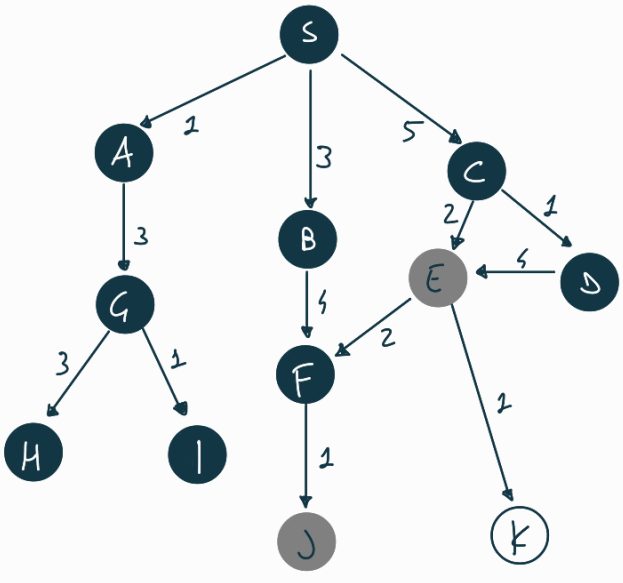
PUSH

J(8); K(7+1)

ORDER BASED ON THE PRIORITY

J(8); K(8)





POP

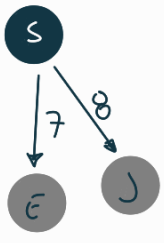
~~H(7)~~; E(7); J(8)

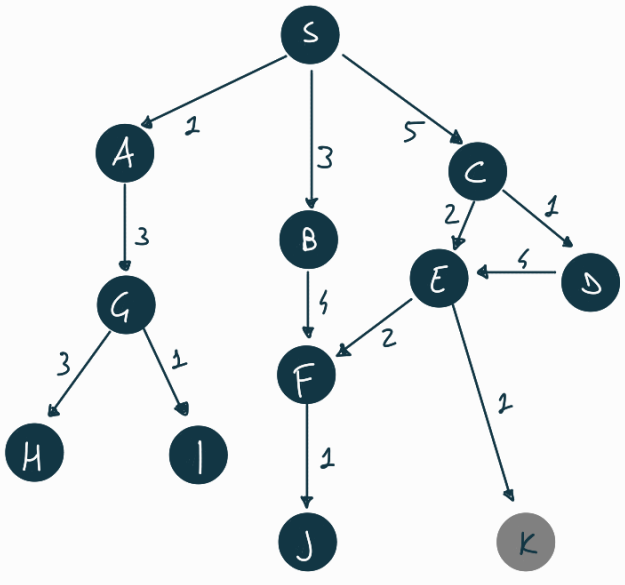
PUSH

E(7); J(8)

ORDER BASED ON THE PRIORITY

E(7); J(8)





POP

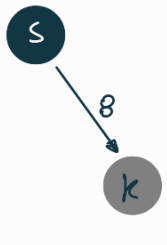
~~J(8)~~; K(8)

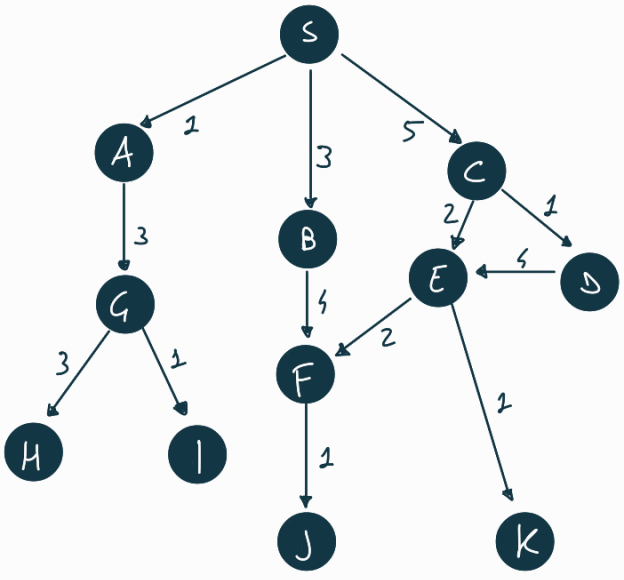
PUSH

K(8)

ORDER BASED ON THE PRIORITY

K(8)





Pop

K (8)

K is the destination → algorithm ends

Depth-first search

The search proceeds exploring first the deepest node of the frontier and then backing up to the second deepest node of the frontier and so on and so ~~for~~ forth, acting in a LIFO way. It is not complete because it could continue to expand nodes always deeper getting stuck in cycles or trying to explore infinite states. It is not optimal because going deeper first, it could miss some shallower solutions. It finds a solution in $O(b^m)$, and it occupies $O(b * m)$ of memory.

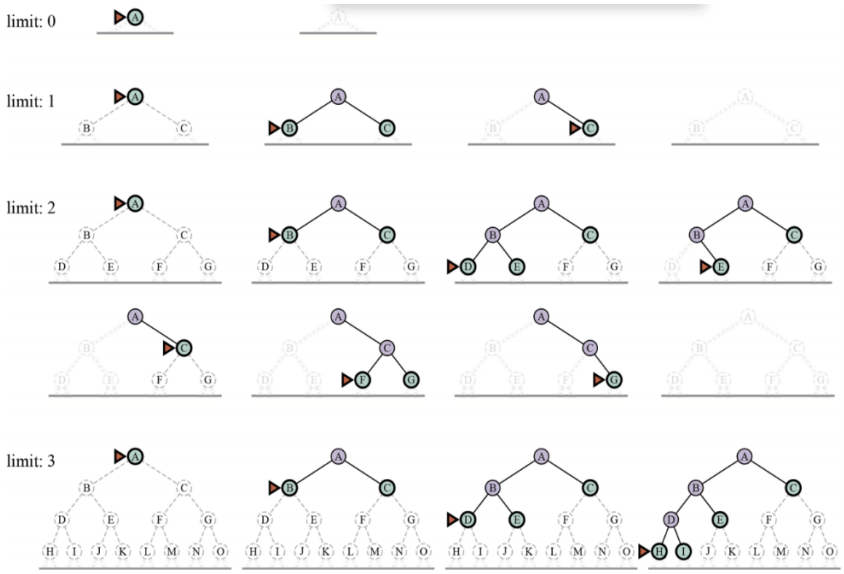
Depth limited search

The search is like depth-first search but there is a limit in the tree's deep. In this way, the searching process cannot continue infinitely. If there is not a solution to a depth minor than the depth limit, the algorithm is not able to find any solution, otherwise, yes; so it is not complete. It is not optimal for the same reason for its incompleteness. It finds a solution in $O(b^d)$, and it occupies $O(b * l)$ of memory.

! complete → ! optimal

Iterative deepening search (IDS)

The search is an iteration of several depth limited searches, starting from l = 1 and increasing it until the shallowest solution is found. It is complete because it finds a solution acting in a similar way of breadth-first search, finding the shallowest solution, with more state visited but with lower memory occupation at the same time. The complexity is $O(b^d)$ for time and $O(b^d)$ for memory occupation.



BFS → spatial complexity = $O(b^d)$, time complexity = $O(b^d)$, complete, optimal
DFS → spatial complexity = $O(b \cdot d)$, time complexity = $O(b^d)$, non complete, non optimal
IDS → spatial complexity = $O(b \cdot d)$, time complexity = $O(b^d)$, complete, optimal

IDS gives us the best of both worlds

Non informed search methods: search methods that do not use specific information about the problem (specific knowledge) - like desirability of a node w.r.t. a goal -. These approaches are not very effective when the state space becomes very large.

Informed search methods: in such cases (large state space) we can take advantages of information about the problem.

Some algorithms make use of an evaluation function $f(n)$ to decide which node to expand first.
Informed search algorithms make use of an heuristic function $h(n)$ which estimate the cost of the cheapest path from the current state (node) to the goal.

Heuristic functions

- Dominance
- Admissibility
- Consistency

Dominance

A heuristic function $h_1(n)$ dominates another heuristic function $h_2(n)$ if:

$$\forall n : h_1(n) \geq h_2(n)$$

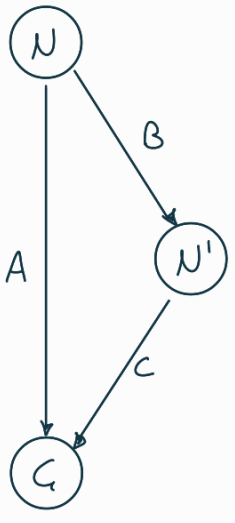
Admissibility → needed for tree search

A heuristic function is admissible if it never overestimates the cost to reach the goal

Consistency → needed for graph search

A heuristic function is consistent if the estimate cost to reach the goal from n is not greater than the cost to reach n' (child of n) plus the estimated cost from n' to the goal

$$h(n) \leq c(n, n') + h(n')$$



$A : h(N)$

$C : h(N')$

$B : c(N, N')$

disuguaglianza triangolare

$$A \leq B + C$$

$$h(N) \leq c(N, N') + h(N')$$

Bidirectional search

An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d. For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state is a successor of in the forward direction, then we need to know that is a successor of in the backward direction. We have a solution when the two frontiers collide.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

informed search algorithms

Heuristic search

Idea: use an evaluation function for each node – estimate of "desirability": expand most unexpanded node desirable. Evaluation function *h(n)* (heuristic) = estimate of cost from *n* to the closest goal.

expand most desirable unexpanded node

Greedy best-first

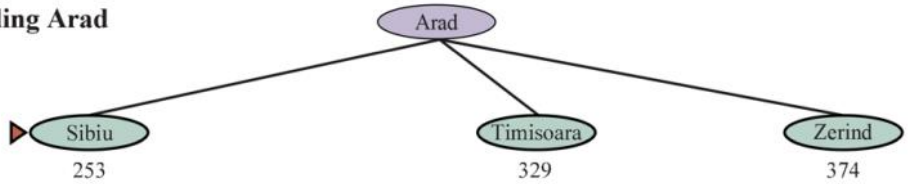
Greedy best-first search is a form of best-first search that expands first the node with the lowest value *h(n)*, ergo the most desirable node —the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. It means that the evaluation function *f(n)=h(n)*. The fringe is a queue sorted in decreasing order of desirability.

It is not complete, can get stuck in loops. Complete in finite space with repeated-state checking. It has time complexity *O(bm)*, but a good heuristic can give dramatic improvement. It has space complexity *O(bm)*—keeps all nodes in memory. It is no optimal.

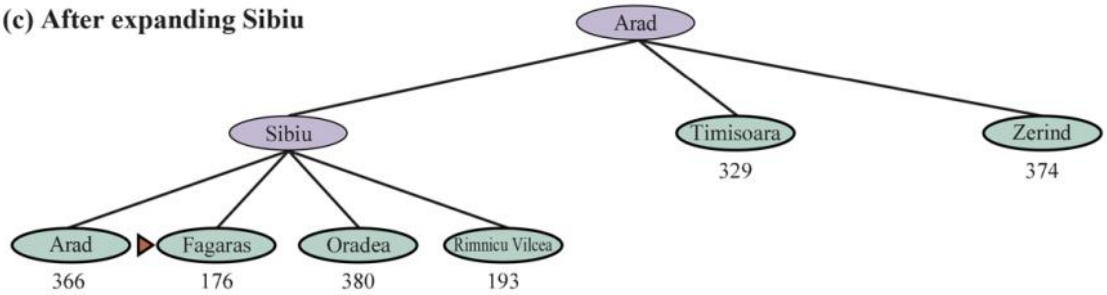
this algorithm uses just the information from the heuristic function it is not complete, not optimal, it has time complexity $O(b^m)$ and spatial complexity $O(b^m)$ where m is the maximum depth of the node to be searched

(a) The initial state

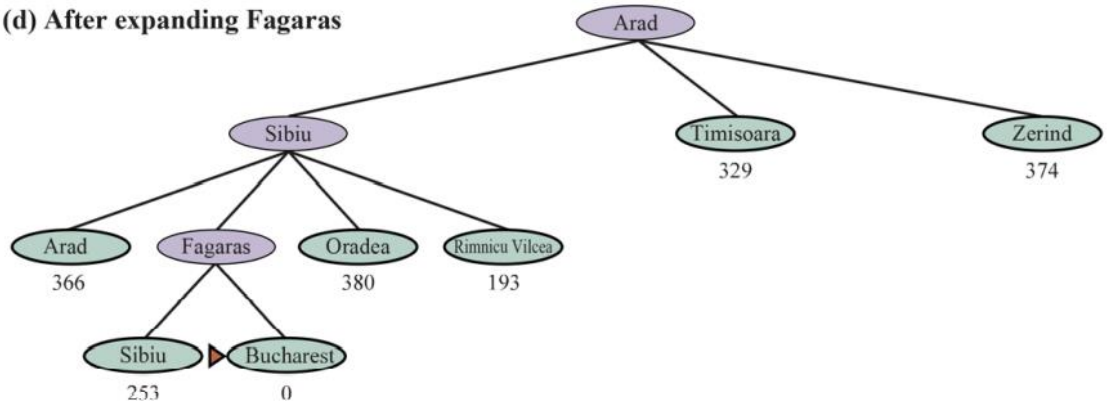
(b) After expanding Arad



(c) After expanding Sibiu



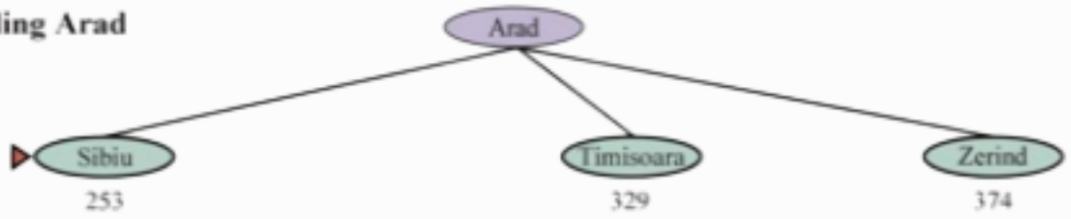
(d) After expanding Fagaras



(a) The initial state

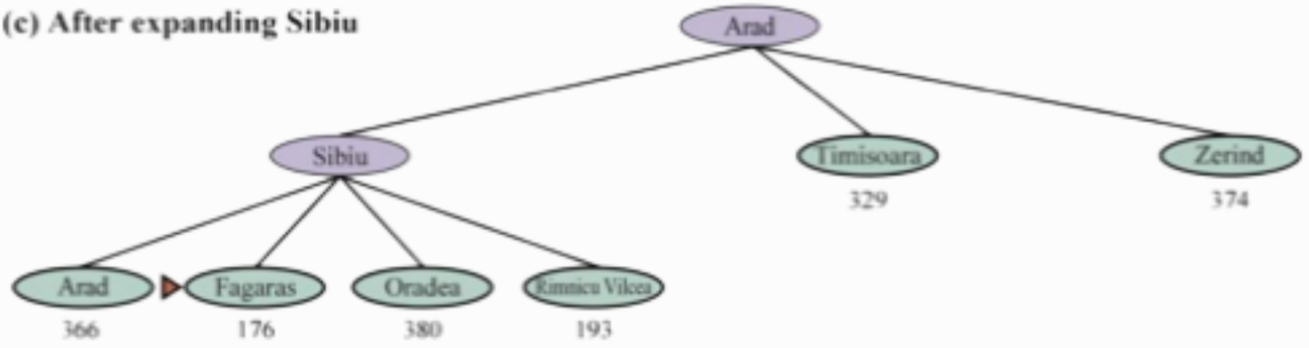
Arad (366)

(b) After expanding Arad



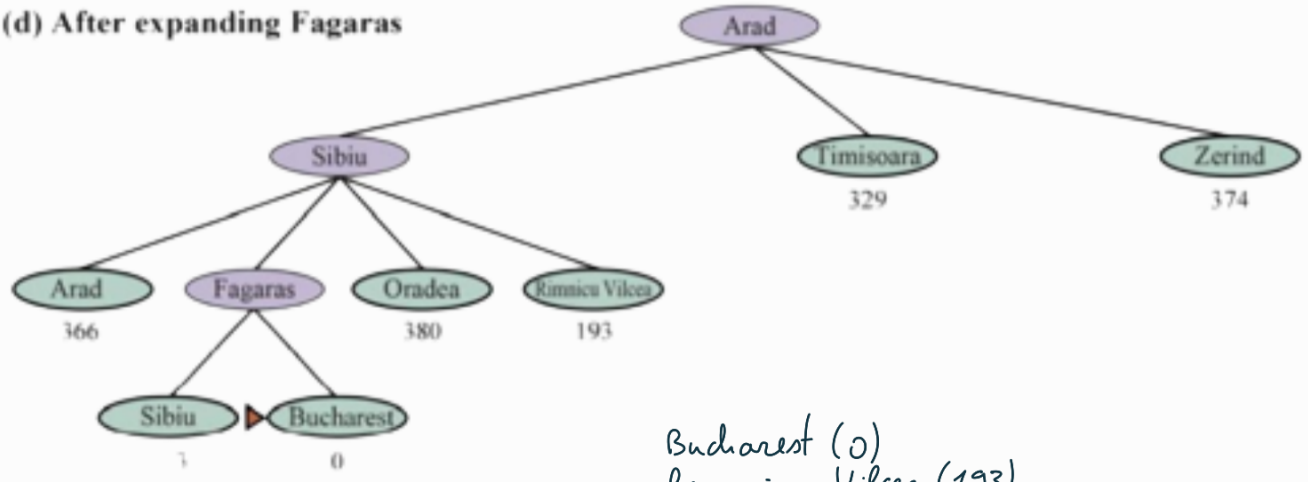
Sibiu (253)
Timisoara (329)
Zerind (374)

(c) After expanding Sibiu



Fagaras (176)
Rimnicu Vilcea (193)
Timisoara (329)
Arad (366)
Zerind (374)
Oradea (380)

(d) After expanding Fagaras



Bucharest (0)
Rimnicu Vilcea (193)
Sibiu (253)
Timisoara (329)
Arad (366)
Zerind (374)
Oradea (380)

Dijkstra's algorithm

Dijkstra's algorithm allows us to calculate the shortest path between a set of points. It works on weighted graphs and can find the shortest paths from a start node and all other nodes on the graph. The shortest path is the sequence of nodes, in the order they are visited, which results in the minimum cost to travel between the start and the end node (cost of a path that connects two nodes being the sum of the weights associated to the edges that forms the path).

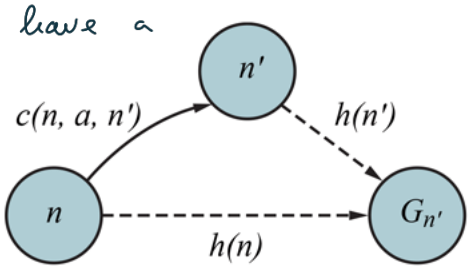
A^* is basically Dijkstra for two nodes and with heuristics.

A* search

The most common informed search algorithm is A* search, a best-first search that uses the evaluation function: $f(n) = g(n) + h(n)$

Where $g(n)$ is the path cost from the initial state to node *n*, and $h(n)$ the estimated cost of the shortest path from *n* to a goal state; so $f(n) =$ *estimated cost of the best path that continues from n to a goal.* NB: a key property is ***admissibility***: an admissible heuristic is one that never overestimates the cost to reach a goal. A slightly stronger property is called consistency. A heuristic $h(n)$ is consistent if, for every node *n* and every successor *n'* of *n* generated by an action *a* we have: $h(n) \leq c(n, a, n') + h(n')$.

Optimal heuristic: a function that does the perfect estimation; if we have a perfect estimation, we have a solution to the problem. We can only have heuristics close to the optimal one.



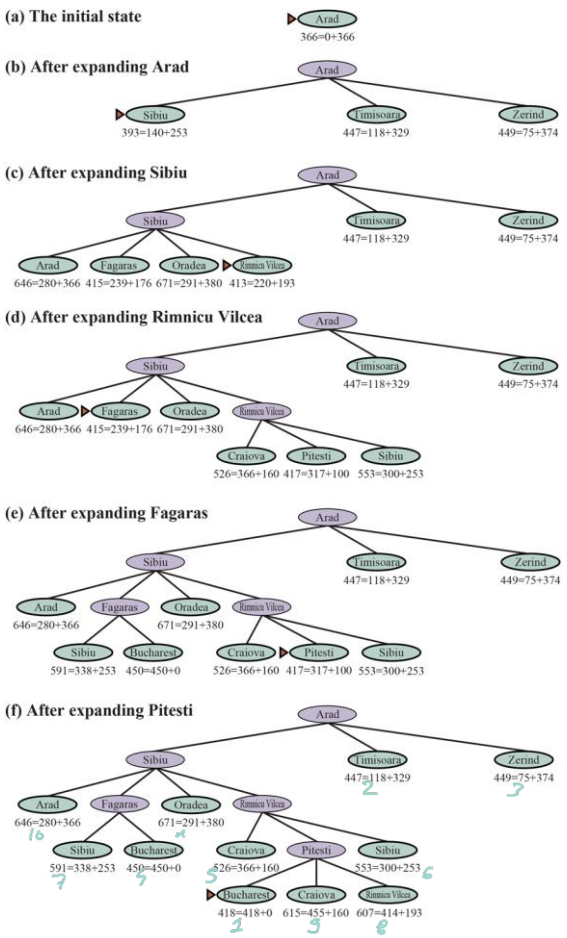
Ordered by best heuristic value

{ 366 Arad }

{ 393 Sibiu, 447 Timisoara, 449 Zerind }

{ 413 Rimnicu Vilcea, 415 Fagaras, 447 Timisoara, 449 Zerind, 646 Arad, 671 Oradea }

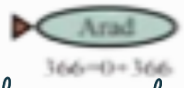
{ 415 Fagaras, 417 Pitesti, 447 Timisoara, 449 Zerind, 526 Craiova, 553 Sibiu, 646 Arad, 671 Oradea }



{ 417 Pitesti, 447 Timisoara, 449 Zerind, 450 Bucharest, 526 Craiova, 553 Sibiu, 591 Sibiu, 646 Arad, 671 Oradea }

{ 418 Bucharest, 447 Timisoara, 449 Zerind, 450 Bucharest, 526 Craiova, 553 Sibiu, 591 Sibiu, 607 Rimnicu Vilcea, 615 Craiova, 646 Arad, 671 Oradea }

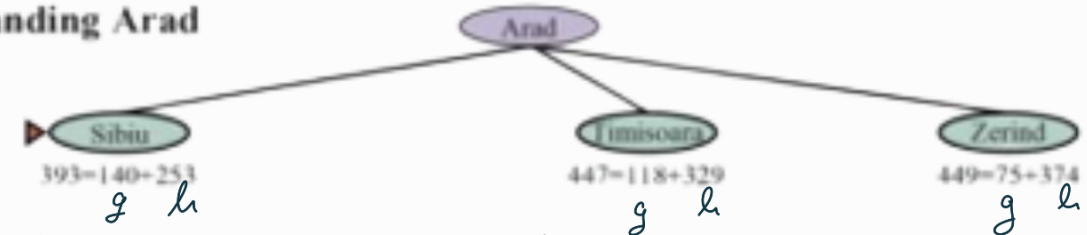
(a) The initial state



$f = g + h$

[Arad (366)]

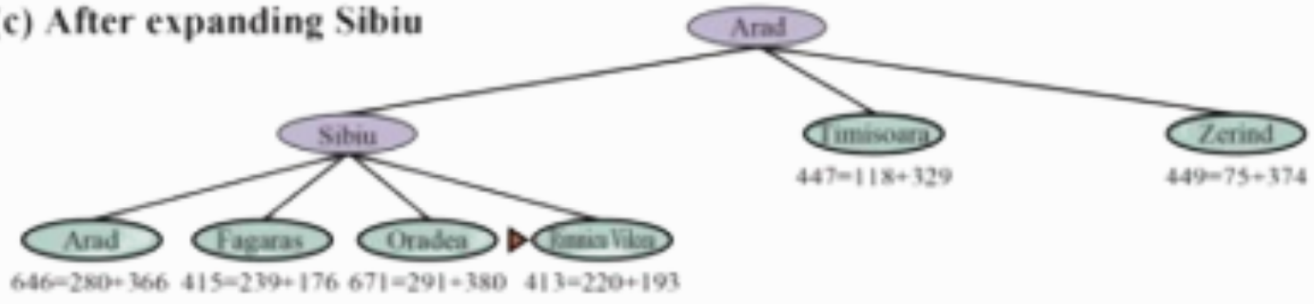
(b) After expanding Arad



g h g h g h

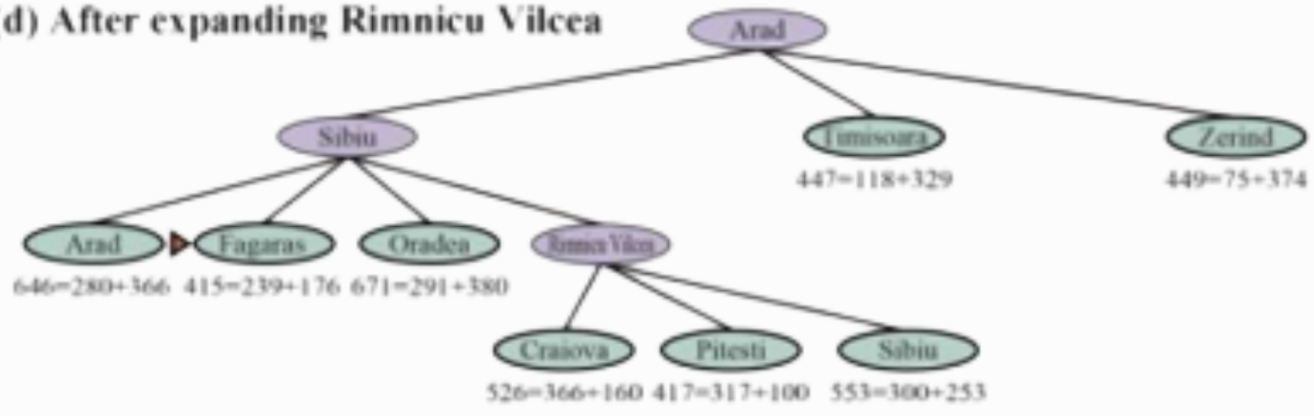
[Sibiu (393), Timisoara (447), Zerind (449)]

(c) After expanding Sibiu



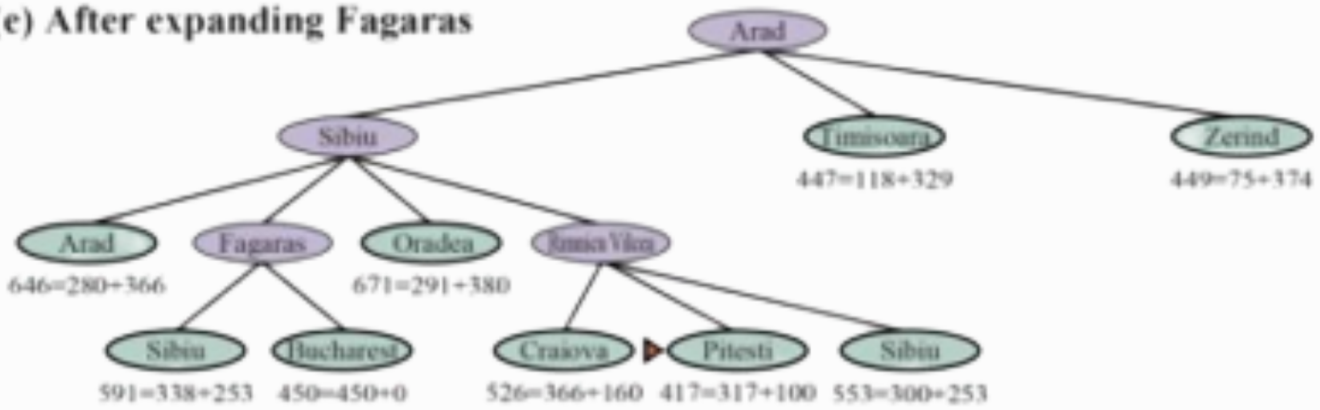
[Rimnicu Vilcea (413), Fagaras (415), Timisoara (447), Zerind (449), Arad (646), Oradea (671)]

(d) After expanding Rimnicu Vilcea



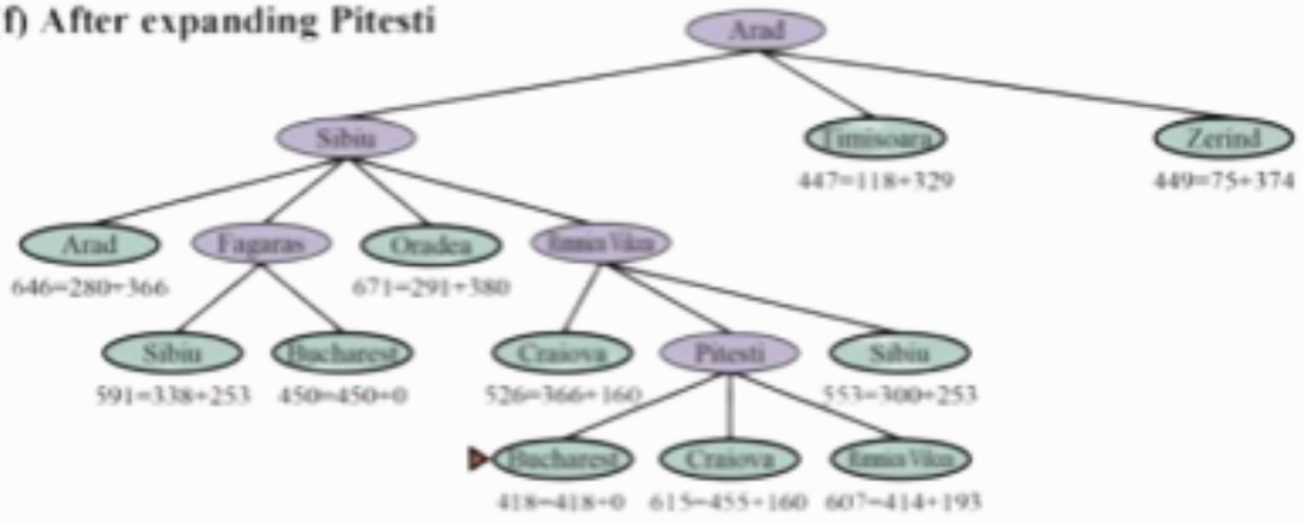
[Fagaras (415), Pitesti (417), Timisoara (447), Zerind (449), Craiova (526), Sibiu (553), Arad (646), Oradea (671)]

(e) After expanding Fagaras



[Pitesti (417), Timisoara (447), Zerind (449), Bucharest (450), Craiova (526), Sibiu (553), Sibiu (591), Arad (646), Oradea (671)]

(f) After expanding Pitesti



[Bucharest (418), Timisoara (447), Zerind (449), Bucharest (450), Craiova (526), Sibiu (553), Sibiu (591), Rimnicu Vilcea (607), Craiova (615), Arad (646), Oradea (671)]

Pseudocode:

```
initialize open list   open_list
initialize closed list   closed_list

push the source to the open_list

while (! open_list.is_empty())

    node = get node with lower f value in open_list
    remove node from open_list
    put node in closed_list

    for each successor n_s of the node

        if n_s is the goal
            return the sequence

        else if n_s is not in the closed_list

            update f = g + h  of n_s

            if n_s is not on the open_list
                add it to the open_list
            else
                check to see if this path to the
                node is better than the one in the list;
                in this case, update the parent of the
                one in the list.
```

Properties of A*

- Complete - Yes, unless there are infinitely many nodes with $f \leq f(G)$
- Time - Exponential in [relative error in $h \times length\ of\ soln.$]
- Space - Keeps all nodes in memory
- Optimal - Yes—cannot expand $fi + 1$ until fi is finished
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n) = C^*$
  - A* expands no nodes with $f(n) > C^*$

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem.

Iterative-deepening A* (IDA*)

IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory. In IDA* the cut-off is the $f - cost\ (g + n)$; at each iteration, the cutoff value is the smallest $f - cost$ of any node that exceeded the cutoff on the previous iteration. In other words, each iteration exhaustively searches an $f - contour$, finds a node just beyond that contour, and uses that node's -cost as the next contour.

Recursive best-first search (RBFS)

RBFS resembles a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f_limit variable to keep track of the $f - value$ of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the $f - value$ of each node along the path with a **backed-up value**—the best $f - value$ of its children. In this way, RBFS remembers the $f - value$ of the best leaf in the forgotten subtree and can therefore decide whether it's worth re-expanding the subtree at some later time. RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. These mind changes occur because every time the current best path is extended, its $f - value$ is likely to increase —h is usually less optimistic for nodes closer to a goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many re-expansions of forgotten nodes to recreate the best path and extend it one more node.

Simplified Memory-Bounded A* (SMA*)

The problem with IDA* and RBFS is that they use too little memory:

- in IDA* at each iteration only the current cost limit for f is kept
- in RBFS the cost of the nodes in the depth first search are recorded.

SMA* instead can use all the available memory for the search. Idea: better remember a node that re-generate it when needed. When a new node must be generated the node with the highest f is discarded (forgotten node), while keeping its cost f in the parent node. A discarded node will be re-generated only when all other paths are worse than the forgotten node.

Properties of SMA*

- It uses all the available memory
- It avoids repeated states until the available memory is exhausted.
- Complete – if the available memory allows the shallowest solution path to be recorded.
- Optimal – if the available memory allows the shallowest optimal solution path to be recorded. Otherwise it returns the best solution attainable with the available memory.
- Optimal Efficient - if the available memory can store the whole search.

Local search

Local search algorithms operate by searching from a start state to neighbouring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages: (1) they use very little memory, and (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable. Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

Hill Climbing

It keeps track of one current state and on each iteration moves to the neighbouring state with highest value—that is, it heads in the direction that provides the steepest ascent. It terminates when it reaches a "peak" where no neighbour has a higher value. Hill climbing does not look ahead beyond the immediate neighbours of the current state. Hill climbing is sometimes called greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Hill climbing can make rapid progress toward a solution because it is usually quite easy to improve a bad state. Unfortunately, hill climbing can get stuck for any of the following reasons:

- **LOCAL MAXIMA**: A local maximum is a peak that is higher than each of its neighbouring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go.
- **RIDGES**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **PLATEAUS:** A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder**, from which progress is possible.

In each case, the algorithm reaches a point at which no progress is being made. There are some possible solutions:

- Side moves with a limit on the maximum number #.
- Stochastic Hill Climbing: random moves (choosing among the uphill successors).
- First choice: random generation of successors taking the first one uphill.
- Random restart.

Success is strongly related to the "shape" of the state space

Simulated annealing

In general

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum. In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum but will be extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing is such an algorithm. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then

gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a very bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum—perhaps into a deeper local minimum, where it will spend more time. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature). It is similar to Hill Climbing Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted.

Local beam search

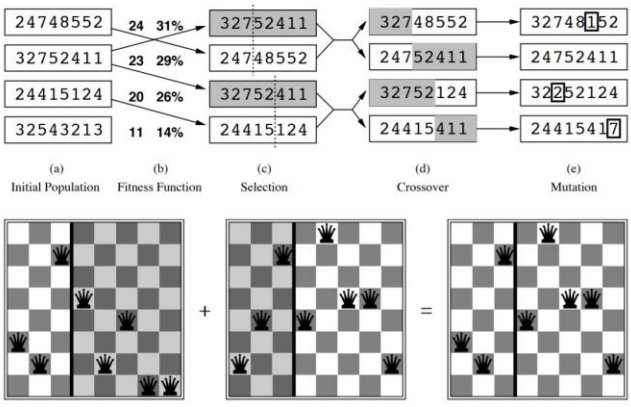
Keeping just one node in memory might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. Problem: too quick convergence in the same region of the search space; the stochastic beam search randomly chooses k successors weighting more the most promising ones.

Genetic Algorithms

Idea: organisms evolve; those adaptable to the environment survive and reproduce, others die (Darwin).

1) initial population: individuals or chromosomes
2) selection: by fitness function
3) reproduction: crossover
4) reproduction: mutation

Search in the space of individuals. Steepest ascent hill-climbing since little genetic alterations are performed on selected individuals.
To deploy Genetic Algorithms, we must define the previous 4 points.



Constraint satisfaction problems (CSPs)

The search algorithms consider each state in an atomic way (indivisible, a black box with no internal structure). Now we break open the black box by using a factored representation for each state: a set of *variables*, each of which has a *value*. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a *constraint satisfaction problem*, or *CSP*.

CSP:
A CSP is a triple $< X, D, Ci >$, where:
X is a set of variables, $\{X1, \dots, Xn\}$.
D is a set of domains, $\{D1, \dots, Dn\}$, one for each variable.
C is a set of constraints that specify allowable combinations of values. Each constraint Cj consists of a pair $< scope, rel >$, where $scope$ is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on.

State is defined by variables Xi with values from domain Di
Goal test is a set of constraints specifying allowable combinations of values vi
for subsets of variables.
Solution is an assignment $\{Xi = vj, \dots\}$ that does not violate the constraints.
Key feature: general-purpose algorithms with more power than standard search algorithms.
Different variables have different domains:
Infinite domains can be reduced to finite by putting an upper bound to values. Unary constraints involve a single variable, e.g., SA 6= green. Binary constraints involve pairs of variables, e.g., SA 6= W A.

Binary CSP: each constraint relates at most two variables.
Constraint graph: nodes are variables, arcs show constraints.
General-purpose CSP algorithms use the graph structure
to speed up search.

Classic approach/Standard formulation

Straightforward approach: states are defined by the values assigned so far.
Initial state: the empty assignment, "{ }"
Successor function: assign value to unassigned variable that does not conflict with current assignment. It fails if no legal assignments (not fixable!). Goal test: the current assignment is complete.

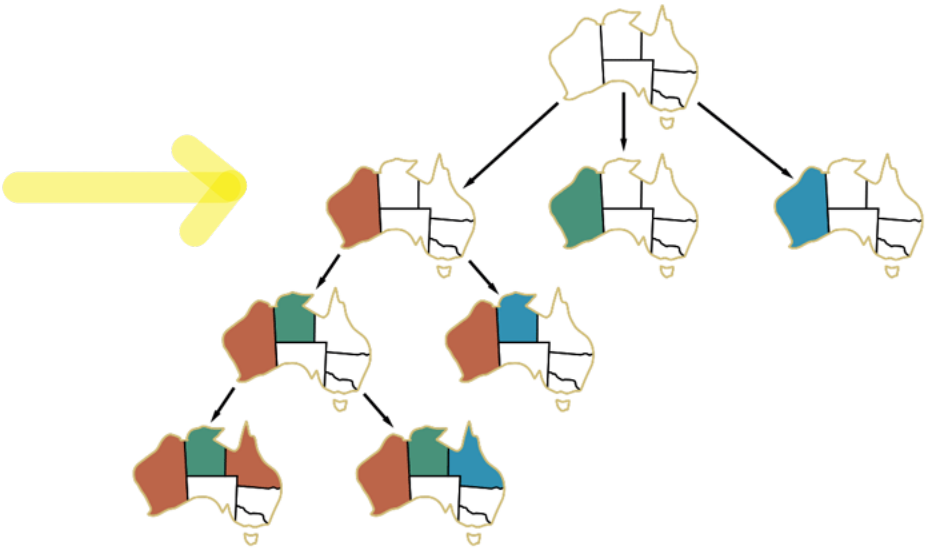
Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Example: Map-Coloring contd.

Solutions are assignments satisfying all constraints, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}$

Backtracking search

Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we need to search for a solution. *Backtracking search* keep in count the CSP's commutative property, so we need only to consider a single variable at each node in the search tree. At each level of the tree we have to choose which variable we will deal with, but we never have to backtrack over that choice. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure.



General-purpose methods can give huge gains in speed:

- *Minimum remaining values (MRV)*: choose the variable with the fewest legal values, it picks a variable that is most likely to cause a failure soon.
- *Degree heuristic*: choose the variable with the most constraints on remaining variables. It attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.
- Given a variable, choose the *least constraining value*: the one that rules out the fewest values in the remaining variables (scegli un valore già assegnato in modo tale da lasciarne molti liberi non assegnati).

Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values. Forward checking propagates information from assigned to unassigned variables but do not provide early detection for all failures.

Search Summary

Author: **Nicolò Brandizzi**

Contributors:

DIAG
Sapienza
October 2018

Contents

Abstract

This is **free** material! You should not spend money on it.
This notes are about the *Search* part taught by professor Daniele Nardi in the Artificial Intelligence class. Everyone is welcome to contribute to this notes in any relevant form, just ask for a pull request and be patient. Remember to add your name under the contributors list in the title page when submitting some changes (if you feel like it).

1 Intro

Some stuff you should remember:

- **Completeness**: The algorithm is guaranteed to find a solution when there is one.
- **Optimality**: The strategy found is optimal.
- **Time Complexity**: The time spent to find a solution
- **Space Complexity** : The memory needed to perform the search.
- **Branching factor** [b]: the maximum number of successors of any node.
- **Depth** [d]: the depth of the shallowest goal node.
- **[m]** : The maximum length of any path in the state space.
- Soundness: An algorithm is sound if it does not yield wrong results

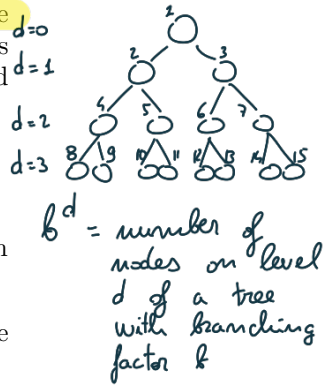
2 Uninformed Search

Only use information available in the problem definition.

Breadth-first It expands the root node first and then all it's successors, and then it's successor and so on. It can be implemented using a *FIFO* queue where the goal condition is applied to each node when it is **generated**.
Proprieties:

- Complete if $d < \infty$
- Optimal if the path cost is a non-decreasing function of the depth (all action have same cost)
- Time and Memory complexities are $O(b^{d+1})$ → number of nodes in a complete tree with branching factor b and depth d

Uniform-cost Expands the node with the lowest path cost $[g(n)]$. It can be implemented using a **Priority Queue** ordered by $g(n)$. The goal condition is applied to the node when this one is selected for *expansion* because the generated one can be a sub-optimal path.
Proprieties:



- Complete if $g(n) > \epsilon$
- Always Optimal since it expands the nodes in order of their optimal path cost
- Time and Memory complexities are $O(b^{C*/\epsilon})$, where $C*$ is the cost of the optimal solution

Depth-first Expands the deepest node in the frontier, using a LIFO queue.

- Complete for *graph version* if space is finite, not complete for *tree version*
- Not Optimal
- Time Complexity is $O(b^m)$ where m is the maximum depth of any node
- Space Complexity is $O(b \cdot m)$

Depth-limited Is the same as *depth-first* but there is a limit for the maximum depth l.

- Complete for $l \geq d$
- Optimal for $l \leq d$
- Time Complexity is $O(b^l)$
- Space Complexity is $O(b \cdot l)$

Iterative deepening depth-first Same as *Depth-limited* but gradually increases l until we have $l = d$.

- Complete when path cost is non decreasing.
- Space Complexity is $O(b \cdot d)$
- Time Complexity is $O(b^d)$
- Optimal if the path cost is a non-decreasing function of the depth (all action have same cost)

Search Direction Can be

- **Forward**: or data driven
- **Backward**: or goal driven
- **Bidirectional**: Uses two simultaneous searches, one from start to goal and one from goal to start, hoping the searches will meet in the middle. It is less intensive since $b^{d/2} + b^{d/2} << b^d$.

Non informed search methods : search methods that do not use specific information about the problem (specific knowledge); these approaches are not very effective when the state space becomes very large.

In such cases you can take advantage of information about the problem

3 Informed Search

Some algorithm make use of an evaluation function $[f(n)]$ to decide which node to expand first (lowest). This information can be joint with an heuristic function $[h(n)]$ which **estimate the cost of the cheapest path from the current state (node n) to the goal**.

3.1 Heuristic functions

Dominance A heuristic function $h_1(n)$ **dominate** another $h_2(n)$ if

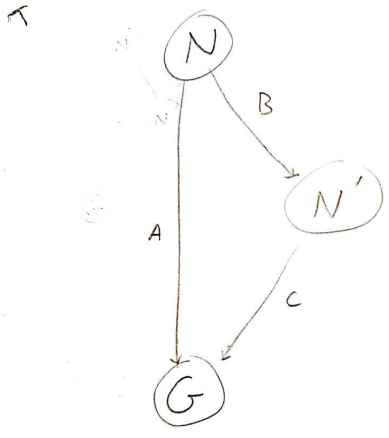
$$\forall n : h_1(n) \geq h_2(n)$$

This leads to the following propriety: **the number of nodes expanded by A* with a dominant h is always less**.

Admissibility For tree search the heuristic function needs to be **admissible**, that is the function never overestimate the cost to reach the goal. For example the path to get from Rome to Madrid being a straight line (euclidean distance) rather than following the roads and avoiding the sea will be an underestimation of the real path.

Consistency For graph search the heuristic function must be **consistence** [1], i.e. the estimate cost to reach the goal from n is not grater than the cost to reach the node n' (child of n) plus the estimated cost from n' to the goal[2]

$$h(n) \leq c(n, n') + h(n')$$



A: h(N)
C: h(N')
B: c(N, @, N')
↳ ACTION

[1] Every consistent heuristic is also admissible
[2] Also known as triangle inequality

Relaxed problem The optimal solution of a relaxed problem is not grater than the optimal solution of the real one. So you can derive a heuristic function from the relaxed problem and apply it to the real one.

Combination Given multiple heuristics $h_1(n), h_2(n), ..., h_m(n)$ you can combine them into a heuristic function which dominates all of them:

$$\forall n : h(n) = max(h_1(n), h_2(n), ..., h_m(n))$$

Effective Brancing Factor [EBF] Given an algorithm whose looking for a goal node and two heuristics associated with it, $h_1(n), h_2(n)$. This algorithm expands $N_1 = 17$ nodes before reaching the solution with the first heuristic and $N_2 = 14$ with the second. Moreover the expansion tree has a brancing factor [3] of $d = 3$. To measure the effectiveness of a heuristic we estimate the **effective brancing factor** [EBF]:

$$N + 1 = 1 + EBF + EBF^2 + ... + EBF^d$$

For our example we have:

$$17 + 1 = 1 + EBF_1 + EBF_1^2 + EBF_1^3 \rightarrow EBF_1 \approx 2.165$$

$$14 + 1 = 1 + EBF_1 + EBF_1^2 + EBF_1^3 \rightarrow EBF_1 = 2$$

The closer EBF is to one the better the heuristic is.

Greedy best-first search This algorithm uses just the information from the heuristic function. The proprieties are the following:

- **Not complete**: can get stuck into loops, can be complete only in finite state with checking.
- **Not Optimal**
- **Time** $O(b^m)$ where m is the maximum depth of the node and b is the branching factor.
- **Space** $O(b^m)$, keeps every node in memory.

3.2 A*

If we combine the information given by the heuristic function, i.e. the cost to reach the goal from the current node, **h(n)** and the cost to reach the node **g(n)**, we get an **estimated cost of the cheapest solution through the node n**

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

Optimality of A* Proving that A* is optimal if h(n) is consistent is fairly easy, we just need to use the formulation of consistency:

$$f(n') = g(n') + h(n') = g(n') + c(n, n') + h(n') \geq g(n) + h(n) = f(n)$$

[4] Moreover the sequence of nodes expanded by A* is in non-decreasing order, hence the first node must be an optimal solution because f is the true cost for the starting point $(h(n_s) = 0)$

[3] The number of successors generated by a given node.
[4] Guarantees that f is not decreasing (monotonic).

Proprieties of A*

goal

- **Is complete** unless there are infinitely many nodes with $f \leq f(G)$
- **Is Optimal**
- **Time is exponential**
- **Space**, keeps every node in memory

3.3 Other implementation of A*

The A* algorithm has some limitations, the most important is the exponential use of the memory (we keep every node in memory).

Iterative Deepening A* [IDA*] The cutoff value is the value of $f(n) = g(n) + h(n)$ rather than the three depth. This behave well for unit cost path finding problems, and keeps a linear memory consumption. But has the same problem of the uniform-cost search

Recursive Best First Search [RBFS] It goes down a node recusivley until the *f-value* increases above the alternative nodes. When this happens it rewinds back to the original node storing the *f-value* of its children. Each rewind and change of mind is an IDA* iteration.
Is has **O(bd)** complexity for space, but its time complexity is not easy to characterize, moreover it uses *too few memory* (only the *f-values* of nodes are stored) thus it can expand a path multiple times.

Simple Memory Bounded A* [SMA*] It behaves just like A*, but when its memory is full it drop worst leaf node [5] propagating its *f-value* up to its parent. When there are same *f-valued* nodes it always expand the newest and deletes the oldest. This algorithm is always complete if the depth (d) of the shallowest solution is in memory size reach.
Proprieties:

- Uses all available memory
- It avoids to repeat states until the memory is full
- It is complete and optimal if d can fit in memory

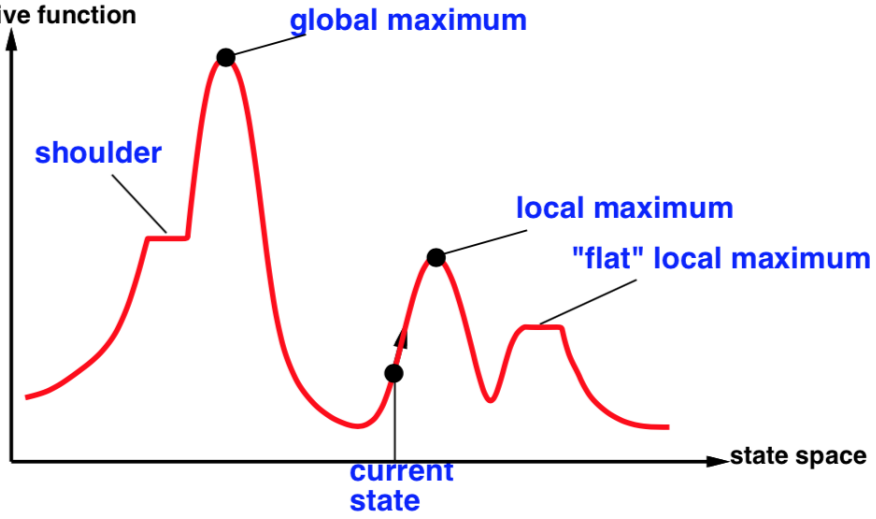
[5] The one with the highest *f-value*.

We look at the search problem from a completely different perspective: technique applicable only when we are not interested in the path leading to the solution; we are interested only in the solution itself.
→ n queen problem
Before → 8 puzzle → we care about the path

4 Local Search Algorithms

Sometimes we don't know the entirety of the state space or we just don't care of the path to reach the goal. For this kind of problem we adopt the local search algorithm whose objective is to gradually optimize a state.

start at a point (state) and modify it step by step in order to find a solution. To find the solution we try to maximize/minimize a function (metrics)
→ fitness function if we maximize



4.1 Hill-climbing search

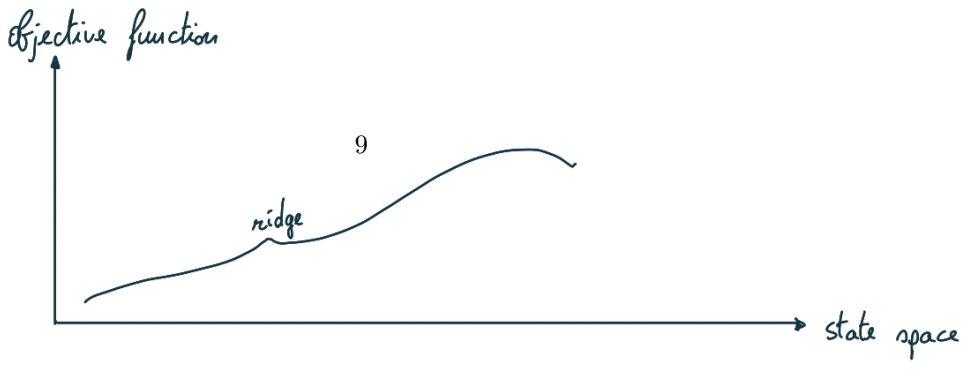
Hill climbing is a simple loop which moves in the direction of the increasing value [6]. Since this algorithm is a **greedy local search** it grabs a good neighborhood without thinking ahead, for this reason it can get stuck in the following cases:

- **Local Maxima**: it won't come down from the maximum (even if it's only local)
- **Ridges** : are a sequence of local maxima which are difficult to navigate [7].
- **Plateaux** : a flat surface which can be either a local minimum or a shoulder.

Stochastic hill-climbing Chooses a randomly from the possible uphill moves, it is slower but can find better solutions.

First choice hill-climbing Implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state

[6] The value is given by an heuristic function.
[7] Look it up on google.



Random restart hill-climbing Randomly generates initial states until a goal is reached. If each hill-climbing has a probability p of succeed then the expected number of restarts required will be $\frac{1}{p}$.

4.2 Simulate annealing

It is a midway between the hill-climbing algorithm and a randomize search [8]. The algorithm picks a random moves every time, if the moves decreases the total cost then it is accepted, on the other hand it has a probability p to be accepted. The probability p depends on two factor:

- The **schedule T** which is a sort of time measure from the start of the algorithm, the more time has passed the lower p will be
- The **badness** δE of the choice to be accepted, i.e. how much worst is the cost after accepting the choice.

If the schedule is slow enough then the algorithm approaches probability 1 to reach a goal.

4.3 Beam Search

This algorithm keeps track of k random generated states, compares them with each other and picks the k best from their successors. Useful information is passed between the k threads so that unfruitful states are abandoned to prefer best ones.
This can bring the search to be concentrated in a small region of the state space. To fix this there is another implementation called the **stochastic beam search** which chooses k successors at random with a probability which is a function of the increasing value.

beam search ~ genetic algorithms

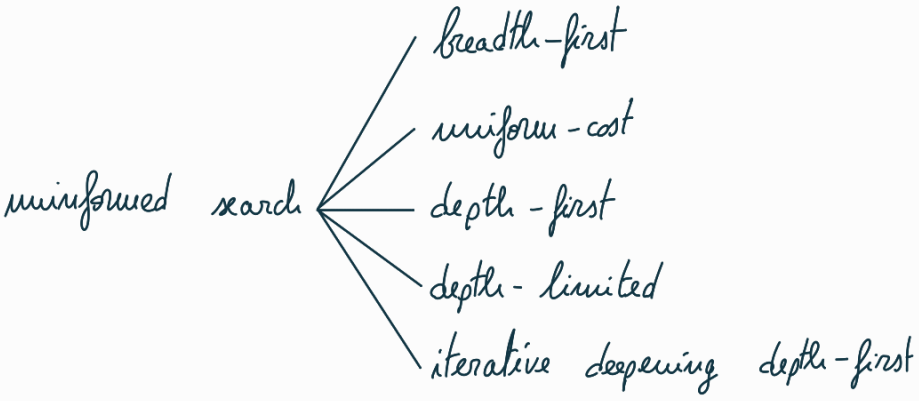
4.4 Genetic Algorithm [GA]

GAs combine uphill tendency with a random exploration of the state space and information exchange between parallel states. The algorithm works like this:

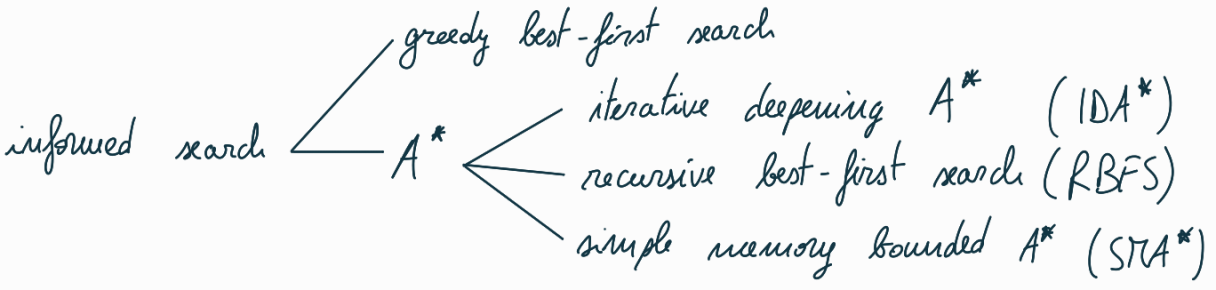
1. At first k random states are generated, called the **population**.
2. The GA rate states using a **fitness function** which return a higher value for better states.
3. A pair of states are chosen according to the rating and a **crossover**[9] point is defined randomly from the states.
4. Finally a random mutation occurs in the children.

[8] Moving to a successor chosen uniformly at random from the set of successors
[9] The crossover usually take large steps at the beginning of the algorithm

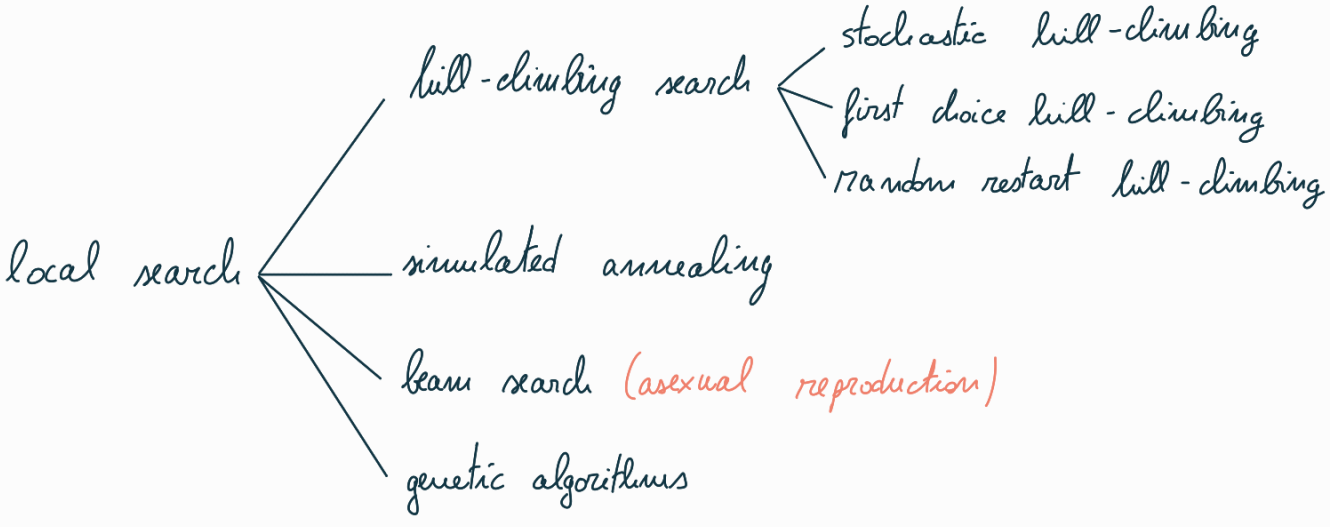
Difference between random restart and beam search:
with random restart we have k independent instances of the same process while in the beam search one process is influenced by the others. With the beam search all the solutions converge in the same spot.



Algorithms have no additional information on the goal node other than the one provided in the problem definition



Algorithms have information on the goal state which helps in more efficient searching. This information is obtained by a function that estimates how close a state is to the goal state.



Local search makes a change in perspective in the search for a solution as compared to other kind of searchs that starts from an initial state and applies operators to reach the goal. Here we don't care about the steps, we only care about the solution: we can start from a candidate solution and find the optimal one. At each iteration we make a small change (perturbation) and then we check how good the newly found solution is with a fitness function (how good the result of this change is). It is not unlikely that the search will stop without finding any solution ⟶ we can only iterate the search starting from a new candidate solution.

they can provide heuristics that do not depend on the problem: they just depend on the structure of the representation.

5 Constraint Satisfaction Problems [CSP]

A CSP has three elements:

- A set of **variables** $X_1, X_2, ..., X_n$, which becomes states when X_i takes its values from D_i.
- A set of **domains** for each variable $D_1, D_2, ..., D_n$
- A set of **constrains** C for legal variable combinations. Which is later called *goal test*.

It is helpful to visualize a CSP as a constraint graph in which every node is a variable and the edges are the constrains. This is done in order to apply *constraint propagation* and delete large portions of search space from the searching.

Key feature: general-purpose algorithms with more power than standard search algorithms

5.1 Variations of CSP

1. initial state: the empty assignment {}
2. successor function: assign value to unassigned variable that does not conflict with current assignment → fail if no legal assignment is possible
3. goal test: the current assignment is complete

The simplest kind of CSP have a discrete finite domain. When the domain is not finite (although is discrete) we must use a constraint language to express the link between variables without enumerating all the legal combinations. As long as the domain is discrete this types of CSP fall under the category of **linear programming** which can be solved in time polynomial to the number of variables.

Arity of constraints

~~**Arytics constrains**~~ There are different kind of constrains:

- **Unary constrains** where a variable is bounded to be a single value ($X = 1$), they can be treated as a domain restriction.
- **Binary constrains** that relates two variables together ($X \neq Y$)
- **Higher order constrains** like $Y \leq X \leq Z$, can be reduced increasing the number of variables and constraints.
- **Global constrains** which involves an arbitrary [10] number of variables (like $AllDiff(X_1, X_2, .., X_n)$)

NP complete: Non deterministic Polynomial complexity

Constraint Optimization Problems [COP] Indicates which solution is preferred, this kind of constrains are not forcefully respected but rather are costly to be neglected.

~~> if for example we have a cost associated to each assignment, we can choose in the end the configuration with the lowest cost → optimization

5.2 Inference in CSPs

We can use **constraint propagation** to reduce the number of legal values for a variable.

Node consistency A single variable is node-consistent if all the values in the variable's domain satisfy the variable's unary constraints. It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

[10] Not necessarily all the variables.

CSP

- backtracking
  - how to choose variable ?
    - minimum remaining variable (MRV) = choose the variable with the fewest legal values
    - degree heuristic = choose the variable with the most constraints on remaining variables
  - how to choose value for a variable ?
    - least constraining value = choose the value that rules out the fewest values in the remaining variables
- forward checking = when making a choice, we keep track of remaining legal values for unassigned variables. It looks only at direct consequences of the assignments
- constraint propagation = forward checking propagates information from assigned to unassigned variables but it doesn't provide early detection for all failures. Sometimes when we remove one of the options from one of the other states, that may have additional consequences.

  → start with direct constraints
  recursively consider the subsequent constraints that can form where a change is made (first step)

for a value assigned to the first variable there must be a consistent choice for the second variable

can either be applied during the search, after each assignment or offline, before the start.

Arc consistency A variable in a CSP is arc-consistent if every value in its domain satisfies the variable's binary constraints. For example given $Y = X^2$ with :

$$\langle (X, Y), [(0, 0), (1, 1), (2, 4), (3, 9)] \rangle$$

If we want to make Y arc-consistent in respect to X we remove from the Y's domain those value which cannot be taken from the equation, so the Y domain will become $[0, 1, 4, 9]$.
The **Ac-3** algorithm is used to propagate arc-consistency in a CSP, it works like this:

1. Initially the queue contains all the arcs in the CSP.
2. AC-3 pops arbitrary arc (X_i, X_j) from the queue.
3. It makes X_i consistent with X_j
4. If this leaves D_i unchanged it continues, else D_i got smaller and the AC-3 gets all the arcs neighborhood of X_i and adds them to the queue
5. If D_i ends up empty the AC-3 return a failure
6. Else it returns the subset of the original domain.

The algorithm lead to faster search at the cost of executing the AC-3 which is $O(cd^3)$ in the worst case scenario.

Path consistency Almost the same as the arc-consistency, in this case we look at a triple of variables X_1, X_2, X_3. A two-variable set X_1, X_3 is path-consistent with respect to a third variable X_2 if, for every assignment $X_1 = a, X_3 = b$ consistent with the constraints on X_1, X_3, there is an assignment to X_2 that satisfies the constraints on X_1, X_2 and X_2, X_3. This is called path consistency because one can think of it as looking at a path from X_1 to X_3 with X_2 in the middle.

K-consistency A CSP is strongly k-consistent if it is k-consistent and is also *(k 1)-consistent, (k 2)-consistent, . . . all the way down to 1-consistent.* Now suppose we have a CSP with n nodes and make it strongly n-consistent, for each X_i we need to find the values in its domain which are consistent with $X_1, X_2, ..., X_{i-1}$. We will find a solution in $O(n^2 d)$ time using exponential space!

Global constrains **Resource constraints** (also called *atmost* constraint), for example the constraint that no more than 10 personnel are assigned in total is written as $AtMost(10, P_1, P_2, P_3, P_4)$ and can be checked estimating the sum. But for large resource-limited problems with integer value we use **bounds propagation**. For example:
F_1 and F_2 are ~~to~~ two flights, for which the planes have capacities 165 and 385, respectively. The initial domains for the numbers of passengers on each flight are then:

$$D_1 = [0, 165] \qquad D_2 = [0, 385]$$

Now suppose we have the additional constraint that the two flights together must carry 420 people: $F_1 + F_2 = 420$, propagating bounds constraints leads to :

$$D_1 = [35, 165] \qquad D_2 = [255, 365]$$

5.3 Backtracking for CSP

We know that CSPs are **commutative** since the assignment of a variable to a value reach the same partial assignment given any order [11]. The *backtracking search* is the same as a depth-search but when a leaf has no legal value left it backtrack the information to its root.

Values and variable order There are some ways to choose the order of variables/values to optimize the time spent building the three.

- **Minimum Remaining Value** [**MRV**] : choosing the variable with the fewest *legal* values.
- **Degree Heuristic** : it attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constrains.
- **Least constraining value**: it prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

Search and Inference There are some techniques to check whenever a choice is better than another. One of them is **forward checking** in which we chose a values for a variable $X_i = c$ and we remove c from all the domains of the variables connected to X_i. This is the case of the constrain *AllDiff*, but the technique works for other examples too.
Another one is **Maintaining Arc Consistency** [**MAC**]: when a value is decided for the variable X_i the algorithm calls AC-3 [5.2] to check arch consistency on all the variables connected with X_i.

Looking Backwards A problem that we may face is the choosing of the right variable X_i which is in direct conflict with the considered variable X_j. There might be cases of an illegal assignment between this two variables but an arbitrary number of other variables $(Y_1, Y_2, ..., Y_k)$ were chosen in the meanwhile. So the previous algorithm will just jump back from $X_j \to Y_K \to Y_{k-1} \to ... \to Y_1 \to X_i$ arriving at the actual problem after k futile approaches. **Backjumping** is used to store a **conflict set** of each variable so that, when an illegal assignment occurs, we can directly jump back to the cause of it.
Remember that every branch pruned by backjumping is also pruned by forward checking.

[11] This brings the possible combination in Sudoku from $n! * d^n$ to d^n