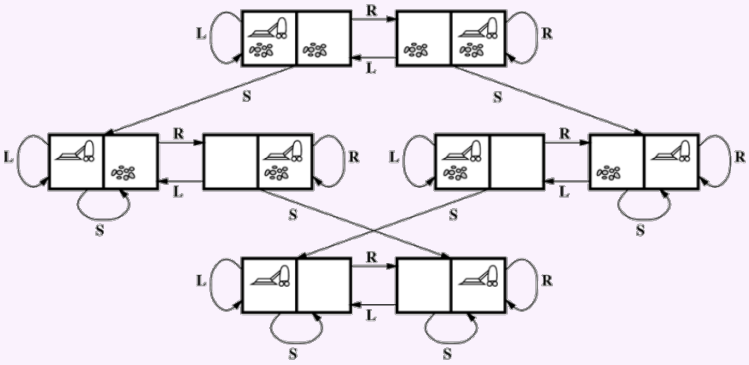
1 Uninformed and Informed Search

1.1 Problem Solving

1.1.1 Key Terminology

State Space / States

A state is a possible situation in our environment - The State Space is a set of all possible states, reachable from the initial state.
Often visualized as a **state-space graph** that displays all reachable states and its transitions.



State-Space Graph of a robot vacuum cleaning

Transition / Action

A Transition describes possible actions to take to get from one state to another. We only count direct transitions between two states (single actions)

Costs

Transitions often differ in different qualities. We add a ”cost” to each action, so we can rate an algorithm on how cost effective it is.

Path

A sequence of states connected by a sequence of actions

Solution

A path that leads from the initial state to a goal state

Optimal Solution

The Solution with the minimal path cost

Problem Components:

1. **State Space and Initial State:**

All possible states and the initial environment as a state

2. **Descriptions of Actions:**

   Function that maps a state to a set of possible actions in this state

3. **Goal Test:**

   Typically a function to test if the current state fulfills the goal

4. **Costs:**

   A cost funciton that maps actions to costs

   An easy way it to add costs of all actions taken

Now that we have defined problems, lets go on to solutions. Most problems can be defined as **planning problems**, in which we start from an initial state and transform it into a desired goal considering future actions and outcomes.

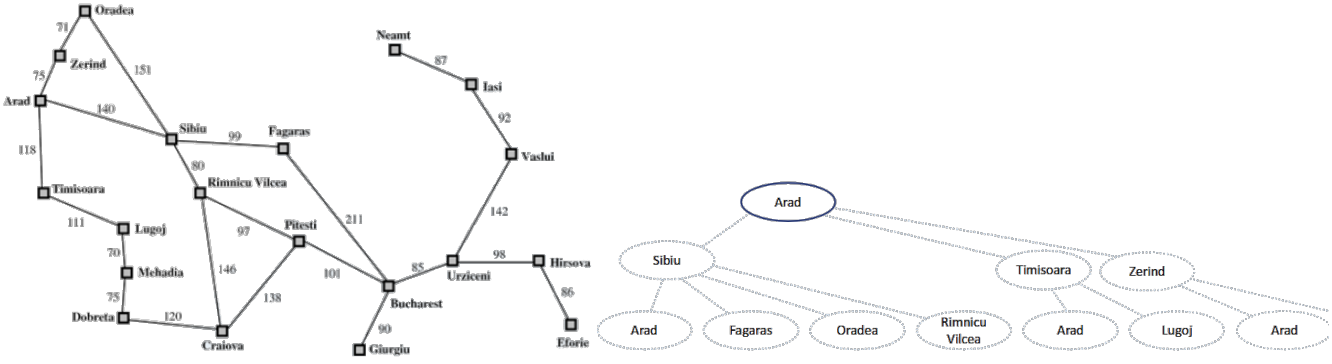
To solve these problems we usually utilize **search** algorithms to find a (optimal) solution in form of a sequence of actions.

1.1.2 Tree Search Algorithms

Hereby we treat the state-space graph as a tree, starting with the initial state. Using this approach we can define iterative or recursive algorithms to search for suitable paths.

Example Tree Search

```
1 Function tree_search(problem, strategy):
2     initialize search tree with initial state of problem;
3     While true do
4         If Node contains goal state then
5             return solution
6         Else If No suitable candidates for expansion then
7             return failure
8         Else
9             Expand node and add resulting nodes to the tree
```



Each Node in the search tree is an entire path in the state-space graph. We construct both on demand - only as much as we need (won't get all solutions)

Node

Describes a part of a tree. Includes **state, parent node, taken action, path cost and depth of the tree**

Fringe

Describes the set of nodes at the end of all visited paths

Depth

Number of levels in the search tree

1.2 Uninformed Search

Definition

The uninformed search strategy only has the problem definition, no additional information.
Some algorithms that utilize uninformed search are:

- Uniform-Cost Search (UCS)
- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Depth-Limited Search (DLS)
- …

1.2.1 Uniform-Cost Search (UCS)

IN UCS each node is associated with a fixed cost, which accumulate over the path. UCS uses the lowest cumulative cost to find a path.

Only works if each step has a positive cost, as otherwise infinite loops would occur

Complexity: $O(b^{1+\lfloor \text{OptimalCost/eps} \rfloor})$ with b being the branching factor (average number of children per node) and eps being the minimal cost for a step

1.2.2 Breadth-First Search

BFS is a special case of UCS, when all costs are equal. It starts at the root and goes through each node of a level before progressing to the next level. BFS stops as soon as it finds a solution, while UCS searches for the 'best' solution by lower cost.

Complexity: $O(b^d)$ with b being the branching factor and d being the depth of the search tree

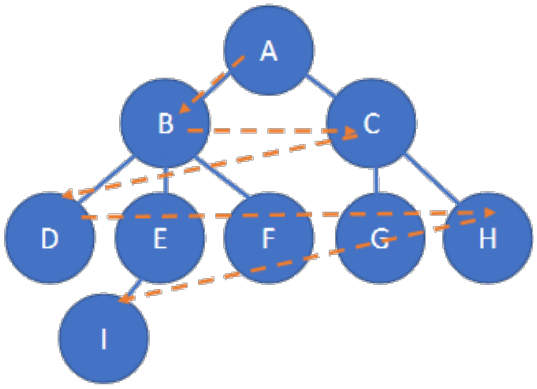
1.2.3 Depth-First Search (DFS)

Depth-First Search starts at the root and continues on one branch until it finds a solution or failure. In case of failure it backtracks to the current parent node and continues on the next branch.

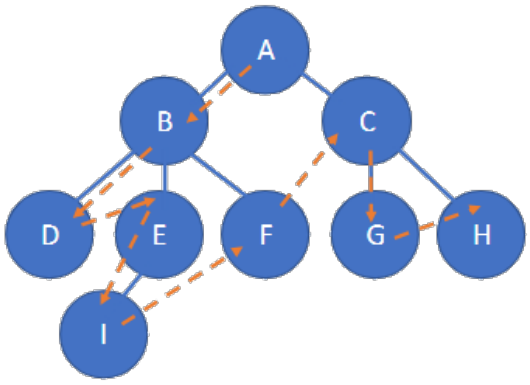
Unlike UCS and BFS DFS is not complete; It fails if the search space is of infinite depth or has loops.

It's also not optimal; More costly solutions may be found before less costly ones.

Complexity: $O(b^m)$ with b being the branching factor and m being the maximum depth of the search tree



BFS: A,B,C,D,E,F,G,H,I



DFS: A,B,D,E,I,F,C,G,H

Criteria	BFS	DFS
Concept	Traversing tree level by level	Traversing tree sub-tree by sub-tree
Data Structure (Queue)	First In First Out (FIFO)	Last In First Out (LIFO)
Time Complexity	O(Vertices + Edges)	O(Vertices + Edges)
Backtracking	No	Yes
Memory	Requires more memory	Less nodes are stored normally (less memory)
Optimality	Yes	Not without modification
Speed	In most cases slower compared to DFS	In most cases faster compared to BFS
When to use	If the target is relatively close to the root node	If the goal state is relatively deep in the tree

Comparison between BFS and DFS algorithms.

1.2.4 Depth-Limited Search (DLS)

DLS is a variant of DFS. Hereby the search is limited to a depth of d. This means that no infinite search can occur, the trade-off being that it might not find all solutions.

Example Depth-Limited Search

```
1  Function depth_limited_search(problem, limit):
2      recursive_dls(make_node(initial_state(problem)), problem, limit)
3  Function recursive_dls(node, problem, limit):
4      cutoff_occured = false;
5      If goal_test(problem, state(node)) then
6          return node;
7      Else If depth(node) >= limit then
8          return cutoff;
9      Else
10         ForEach successor in expand(node, problem) do
11             result = recursive_dls(successor, problem, limit);
12             If result == cutoff then
13                 cutoff_occured = true;
14             Else If result != failure then
15                 return result;
16     If cutoff_occured then
17         return cutoff;
18     Else
19         return failure
```

This, of course, is neither complete nor optimal.

Complexity: $O(b^l)$ with b being the branching factor and l being the maximum depth of the search tree

1.2.5 Iterative Deepening Search (IDS)

IDS is a variant of DLS. It works like DLS, but instead of a fixed depth, it iteratively increases the depth until a solution is found.

Example Iterative Deepening Search

```
1  Function iterative_deepening_search(problem):
2      For depth = 0 to ∞ do
3          result = depth_limited_search(problem, depth);
4          If result != cutoff then
5              return result
```

This change makes it complete and optimal.

Complexity: $O(b^m)$ with b being the branching factor and d being the depth of the search tree. This makes its time complexity equal to BFS. However, the space complexity of IDS is $O(bd)$, which is much better than BFS' $O(b^d)$.

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^d	$b^{1+\lfloor OptCost/eps \rfloor}$	b^m	b^l	b^d
Space	b^d	$b^{1+\lfloor OptCost/eps \rfloor}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Comparison of search strategies.

1.3 Heuristics h

Denotes a "rule of thumb", a rule that may be helpful in solving a problem.
In Tree-Search, a heuristic is a function that estimates the remaining cost from the current node to the goal.
Can go wrong!

Admissible Heuristic

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

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

if $h^*(n)$ is the true cost to reach the goal from node n

Consistent Heuristic

A heuristic is consistent if for every node n and successor n' generated by any action a:

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

if $c(n, a, n')$ is the cost of the action a from n to n'
Thus, a heuristic is consistent if, when going from neighboring nodes, the heuristic difference / step cost never overestimates the actual cost.

Lemmas

- If a heuristic is consistent, it is also admissible.
- If $h(n)$ is consistent, then the values of $f(n)$ on any path are non-decreasing.

1.3.1 Relaxed Problems

A relaxed problem is a problem that has fewer constraints on the actions than the original problem.

The cost of the optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Example: Relaxed Problem as Admissible Heuristic
Consider a grid-based pathfinding problem where certain movements are restricted due to obstacles. A relaxed version of this problem might allow movement through obstacles by ignoring some constraints. The cost of the optimal solution to this relaxed problem, which is typically lower or equal to the original problem's optimal cost, serves as an admissible heuristic for the original problem.

This means that looking for relaxed problems is a good way to find admissible heuristics.

1.3.2 Dominance

A heuristic h_2 dominates h_1 if $h_2(n) \geq h_1(n)$ for all nodes n. (Given that h_1 and h_2 are admissible)

This means that a dominant heuristic is always closer to the optimal heuristic h*, which results in less expansion and thus more efficient search.

1.3.3 Combining Heuristics

If h_1, h_2, ...h_m are admissible heuristics, then $h(n) = max(h_1(n), h_2(n), \ldots, h_m(n))$ is also admissible and dominates both h_1 and h_2.

This is useful if we have multiple admissible non-dominated heuristics and want to combine them to get a better heuristic.

1.4 Informed Search

Definition

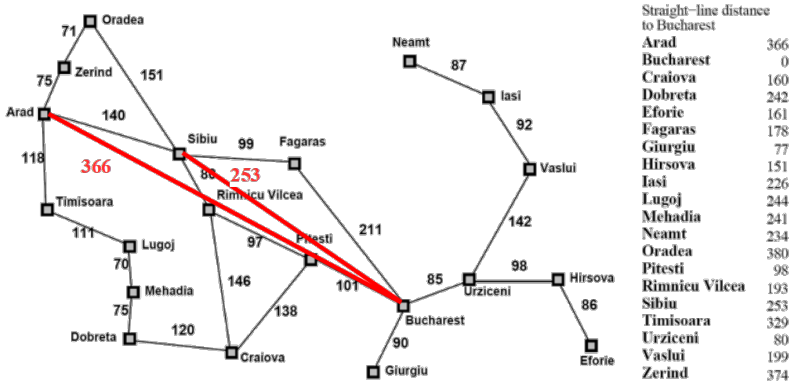
The informed search strategy has additional knowledge about "where" to look for solutions, usually in form of a heuristic function.
Some algorithms that utilize informed search are:

- Greedy Best-First Search
- A* Search
- Memory-Bounded Heuristic Search

1.4.1 Greedy Best-First Search

Greedy Best-First Search (GBFS) is an informed search algorithm that selects the next node to expand based on a heuristic that estimates the cost from the current node to the goal.

One example of a heuristic is the straight-line distance to the goal.



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example of GBFS with a straight-line distance heuristic.

Complexity: The time complexity of Greedy Best-First Search is $O(b^m)$, and its space complexity is also $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree. This is the same as DFS, however due to the heuristic it can be significantly faster.

GBFS is not guaranteed to be optimal or complete, especially if the heuristic function is not admissible or consistent. Its efficiency heavily depends on the quality of the heuristic used.

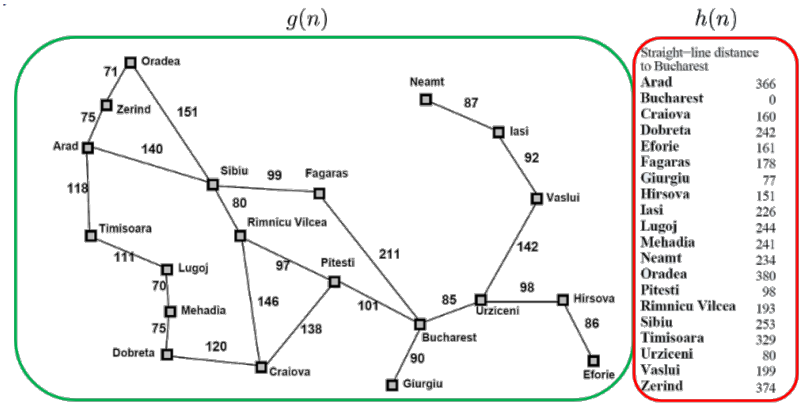
- **Complete?** No, can get stuck in loops.
- **Optimal?** No, does not guarantee an optimal solution as it depends on the heuristic, which may be flawed.

1.4.2 A* Search

Definition

A* Search is an informed search algorithm that combines the strengths of Uniform-Cost Search and Greedy Best-First Search. It searches for the least-cost path to the goal by considering both the cost to reach the current node and an estimated cost to reach the goal.

A* Search tries to avoid paths that are already expensive. It evaluates the complete path cost and the remaining cost to the goal. Its cost function is defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach the current node (root to n) and $h(n)$ (the heuristic) is the estimated cost to reach the goal (n to goal).



Example of A Search with heuristic function.*

Complexity: The time complexity of A* Search is $O(b^m)$, and its space complexity is also $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.

Completeness and Optimality: A* Search is complete, and can be optimal, if the heuristic is admissible.

1.4.3 Alternatives to A*

1. **Iterative-Deepening A* (IDA*)**
   - Like iterative deepening, it explores nodes level by level, but uses A* to evaluate the nodes.
   - Cutoff information is the f-cost (g + h) instead of the depth.
2. **Recursive Best-First Search (RBFS)**
   - Recursive algorithm that mimics best-first search with linear space
   - Keeps track of f-value of best alternative path
   - Path available from any ancestor of the current node and heuristic evaluations are updated with results of successors
3. **(Simple) Memory-Bounded A* ((S)MA*)**
   - Drops the worst leaf node when memory is full
   - Its value will be updated to its parent
   - May be researched later

1.4.4 Graph Search

When traversing a problem, loops can occur. Failure to detect them can turn linear search into exponential search. To avoid this, we can use **graph search**. Hereby we only expand nodes that have not been visited yet.

For example: The graph search version of UCS is **Dijkstra's algorithm**.