CSC384

Introduction to Artificial Intelligence

Sinan Li

2024

CONTENTS

I Lecture Notes **5**

1 **Chapter 1**
Introduction to Artificial Intelligence

1.1 Artificial Intelligence (AI) *7*
1.1.1 Intelligence *7*
1.1.2 The Turing Test *7*
1.1.3 Computational Intelligence *8*
1.2 Rationality *8*
1.3 Subareas of AI *9*
1.3.1 Further Courses in AI *9*
1.4 A Brief History of AI *9*

2 **Chapter 2**
Search

2.1 Search Problems *11*
2.1.1 Formalizing a Problem as a Search Problem *11*
2.1.2 Graphical Representation *13*
2.1.3 Algorithms for Search *14*
2.2 Uninformed Search Algorithms *15*
2.2.1 Uninformed Search Strategies *15*
2.2.2 Breadth-First Search (BFS) *16*
2.2.3 Depth-First Search (DFS) *17*
2.2.4 Depth-Limited Search (DLS) *18*
2.2.5 Iterative Deepening Search (IDS) *19*
2.2.6 Path Checking and Cycle Checking *21*
2.2.7 Uniform-Cost Search (UCS) *23*
2.3 Heuristic Search *25*
2.3.1 Heuristic Search Strategies *25*
2.3.2 Greedy Best-First Search (GBFS) *26*
2.3.3 A* Search *27*
2.3.4 Iterative Deepening A* (IDA*) *30*
2.3.5 Building Heuristic Functions *30*

3 Chapter 3
Game Tree Search

4 Chapter 4
Constraint Satisfaction Problems

5 Chapter 5
Representing and Reasoning under Uncertainty

6 Chapter 6
Symbolic Knowledge Representation and Reasoning

II Tutorials **41**

III Appendices **49**

Bibliography **51**

Index **53**

Part I

Lecture Notes

CHAPTER

1 INTRODUCTION TO ARTIFICIAL INTELLIGENCE

1.1 Artificial Intelligence (AI)

Artificial Intelligence is a branch of computer science utilizing *computational* ideas and examining how we can achieve *intelligent* behavior through computation.

1.1.1 Intelligence

The ability to apply knowledge to manipulate one's environment or to think abstractly as measured by objective criteria (such as tests)

— *Merriam-Webster Dictionary*

Human features that are considered intelligent includes the ability to learn, understand, reason, plan, communicate, and perceive. For example, a human can learn to play a game, understand the rules of the game, reason about the best moves to make, plan a sequence of moves, communicate with other players, and perceive the state of the game.

1.1.2 The Turing Test

The Turing Test is a test of a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human. The test was introduced by Alan Turing in his 1950 paper, *Computing Machinery and Intelligence*, while working at the University of Manchester [1]. Turing proposed that a human evaluator would judge natural language conversations between a human and a machine designed to generate human-like responses. The evaluator would be aware that one of the two partners in conversation is a machine, and all participants would be separated from one another. The conversation would be limited to a text-only channel such as a computer keyboard and screen so that the result would not be dependent on the machine's ability to render words as speech. If the evaluator cannot reliably tell the machine from the human, the machine is said to have passed the test. The test does not check the ability to give the correct answer to questions; it checks how closely the answer resembles typical human answers. The conversation is limited to a single topic chosen by the examiner.

- Turing provided some very persuasive arguments that a system passing the Turing test is *intelligent*.
    - We can only really say it *behaves* like a human

- **No guarantee** that it *thinks* like a human

- The Turing test does not provide much traction on the question of **how to build** an intelligent system.

Why not just simulate human's brain?

- Brains are very good at making rational decisions, but **not perfect**.
- Brains **aren't as modular** as software, so hard to reverse engineer!
- Computers and Humans have quite **different abilities**.
  - *Memory* and *simulation* are key to decision making.
  - *Perceptual tasks* (vision, sound, etc.) are effectively accomplished by architectures related to the way the brain works (deep neural networks).

1.1.3 Computational Intelligence

Artificial Intelligence tries to understand and model *intelligence* as a **computational process**. Thus we try to construct systems whose **computation** achieves or approximates the desired notion of intelligence.

Other areas interested in the study of intelligence lie in other areas or study, e.g., cognitive science which focuses on human intelligence. Such areas are very related, but their central focus tends to be different.

1.2 Rationality

Formally, we can define an **agent** as something that **perceives** and **acts** in an *environment*. An agent can be a *human*, a *robot*, or a *software agent*.

A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. Rationality is distinct from omniscience (knowing everything) and omnipotence (being able to do anything). Rationality maximizes **expected utility**, which is the sum of the utility of each possible outcome of an action weighted by its probability of occurring.

Rationality is measured by the *outcome*, not the *action* itself. It is a precise *mathematical* notion of of what it means to do *the right thing* in any particular circumstance. Provides

- A **precise mechanism** for analyzing and understanding the properties of this ideal behaviour we are trying to achieve.
- A **precise benchmark** against which we can measure the behaviour the systems we build.

Trying/Expectation

Rational action is not always equal to rational decision.

1. We often don't have **full control** or **knowledge** of the world we are interacting with.
2. We usually don't know **precisely** what the **effects** of our actions will be.

In some contexts we can *simplify* the computational task by assuming that we do have full knowledge/-control.

1.3 Subareas of AI

A common misconception is to equate AI with Machine Learning. But AI is much more than that. This course will not focus on Machine Learning, but rather on the other subareas of AI. What we cover is not an exhaustive list of all subareas of AI, but rather a starting point for further exploration.

- **Perception**: vision, speech understanding, etc.
- **Machine Learning**, **Neural Networks**
- **Robotics**
- **Natural Language Processing**
- **Reasoning and Decision Making**
  - **Symbolic Knowledge Representation**
  - **Reasoning** (logical, probabilistic)
  - **Decision Making** (search, planning, decision theory)

1.3.1 Further Courses in AI

- Perception: vision, speech understanding, etc.
  - CSC487H1 "Computational Vision"
  - CSC420H1 "Introduction to Image Understanding"
- Machine Learning, Neural networks
  - CSC311H "Introduction to Machine Learning"
  - CSC412H "Probabilistic Learning and Reasoning"
  - CSC413H1 "Neural Networks and Deep Learning"
- Robotics
  - Engineering courses
- Natural language processing
  - CSC401H1 "Natural Language Computing"
  - CSC485H1 "Computational Linguistics"
- Reasoning and decision making
  - CSC486H1 "Knowledge Representation and Reasoning"

1.4 A Brief History of AI

- 1940-1950: Early days
  - 1943: McCulloch & Pitts: Boolean circuit model of brain
  - 1950: Turing's "Computing Machinery and Intelligence"
- 1950—70: Excitement: Look, Ma, no hands!
  - 1950s: Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine

  - 1956: Dartmouth meeting: "Artificial Intelligence" adopted
  - 1965: Robinson's complete algorithm for logical reasoning
- 1970—Early 2000: Knowledge-based approaches
  - 1969—79: Early development of knowledge-based systems
  - 1980—88: Expert systems industry booms
  - 1988—93: Expert systems industry busts: "AI Winter"
  - 1997: IBM's Deep Blue beats chess grandmaster Garry Kasparov
- Early 2000— present: Statistical approaches
  - Resurgence of probability, focus on uncertainty
  - Agents and learning systems... "AI Spring"
  - 2007: DARPA Urban Challenge – CMU autonomous vehicle drives 55 miles in an urban environment while adhering to traffic hazards and traffic laws.
  - 2016: AlphaGo beats 9-Dan pro Go player Lee Sedol
  - 2017: AlphaGo Zero – learns by playing with itself
  - 2022: Large Language Models (LLM) Chat Generative Pre-trained Transformer, which has been fueled by advances in Neural Net architecture and access to big data.

There are many **unsolved** problems yet... including lots of **legal/ethical** ones.

CHAPTER

2 SEARCH

2.1 Search Problems

Searching is one of the most *fundemental techniques* in AI, underlying sub-module in many AI systems. Ai can *solve* many problems that homansare not good at, and achieve *super-human performance* in many domains (e.g. chess, go, etc.).

- **Benefits**
  - Useful as ageneral algorithmic technique for solving problems, both in AI and in other areas.
  - Outperform humans in some areas (e.g. games).
  - Practical:
    - Many problems don't have specific algorithms for solving them.
    - Useful in approximation (e.g., local search in optimization problems).
  - Some critical aspects of intelligent behaviour, e.g., planning, can be cast as search.
- **Limitations**
  - Only shows how to solve the problem once we have it correctly formulated.

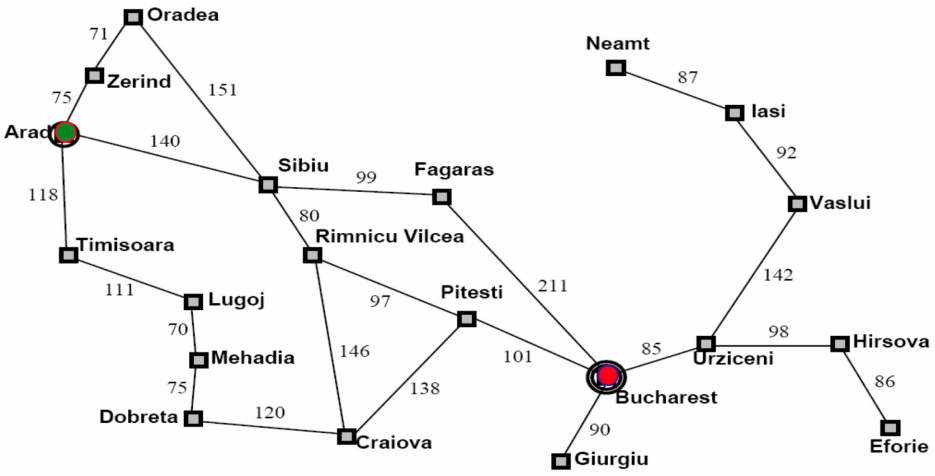
2.1.1 Formalizing a Problem as a Search Problem

- Necessary components
  1. **State Space**: A **state** is a representation of a *configuration* of the problem domain. The state space is the *set of all states* include in our model of the problem.
  2. **Initial State**: The starting configuration.
  3. **Goal State**: The configuration one wants to achieve.
  4. **Actions** (or State Space Transitions): Allowed changed to move from one state to another.
- Optional Ingredients
  - *Costs*: Representing the cost of moving from state to state.

- *Heuristics*: Help guide the search process.

Once a search problem is formalized, there are a number of algorithms one can use to solve it. A **solution** is a *sequence of actions* or moves that can transform the **initial state** into a **goal state**.

Example (Romania Travel)**.** We want to travel in Romania from Arad to Bucharest as fast as possible.



Each state would be a city.

- **State Space**: The set of all cities on the map.
- **Initial State**: Arad.
- **Goal State**: Bucharest.
- **Actions**: Driving between neighbouring cities.

♢

Example (Water Jugs)**.** We have a 3-liter jug and a 4-liter jug. We can fill either jug to the top from a tap, or we can empty either jug onto the ground. We can also pour the contents of one jug into the other until the receiving jug is full or the pouring jug is empty.

Suppose initially the 4-liter jug is full, we want to have exactly 2 liters in the 3-liter jug.

We can use a pair of numbers to represent the state of the system: the amount of water in the 3-liter jug and the amount of water in the 4-liter jug.

- **State Space**: The set of all pairs of numbers (a, b) where a is the amount of water in the 3-liter jug and b is the amount of water in the 4-liter jug.
- **Initial State**: $(0, 4)$.
- **Goal State**: $(2, 0)$, $(2, 1)$, $(2, 2)$, $(2, 3)$, $(2, 4)$.
- **Actions**:
  - Fill the 3-liter jug from the tap.
  - Fill the 4-liter jug from the tap.
  - Empty the 3-liter jug onto the ground.
  - Empty the 4-liter jug onto the ground.
  - Pour the contents of the 3-liter jug into the 4-liter jug.
  - Pour the contents of the 4-liter jug into the 3-liter jug.

Remark

When formalizing a search problem, always consider these questions:

1. Can we reach all states fron any given start state?
2. Will all actions result in a change of state?

   No! Imagine you have $(3, 4)$ and you try to fill the 3-liter jug from the tap. You will still have $(3, 4)$.

♢

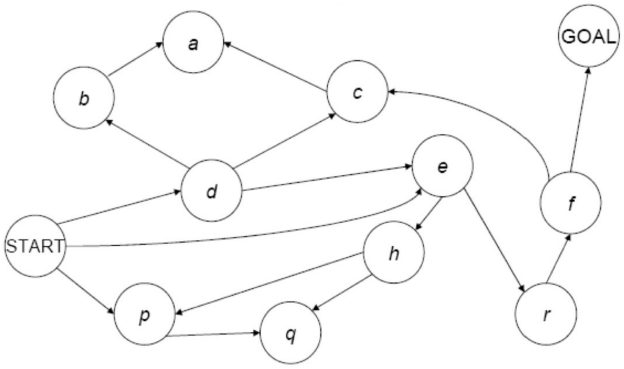
In more complex situations,

- Actions may lead to **multiple states**.

  For example, filpping a coin may lead to heads or tails.
- We may not be **sure of a given state**

  For example, when prize is behind door 1, 2, or 3.
- Such situations require techniques for reasoning under uncertainty: assign probabilities to given outcomes.

2.1.2 Graphical Representation



Assuming a finite search space, the

- **vertices** represent states in the search space; and the
- **edges** represent transitions resulting from actions (or successor functions).

Search Tree

Definition 2.1.1 Search Tree

A **search tree** is a *directed graph* where

- Each node represents a state.
- Each edge represents an action.
- The root node represents the initial state.
- The leaf nodes represent goal states.

A search tree reflects the behaviour of an algorithm as it walks through a search problem. It has two important properties:

- **Solution depth**, denoted d, the depth of the shallowest goal node in the tree.
- **Maximum branching factor**, denoted b, the maximum number of children of any node in the tree.

> **Remark**
> Note that the **same** state may appear **multiple times** in a search tree.

> **Remark**
> It is important to distinguish between **states** from **nodes**.
>
> - A **state** represents a possible configuration of the world.
> - A *node* is a data structure constituting part of a search tree. It includes
>   - a **state** and
>   - the **parent node**,
>   - the **action** that led to this node, and
>   - the **cost** of the path from the initial node to this node.
> - Intuitively speaking, each node corresponds with a path from the initial state to the node's state.
> - Two **different nodes** are allowed to contain the **same world state**.

2.1.3 Algorithms for Search

- **Input**
  - **Initial node**
  - **Successor Function** $S(x)$
    returns the set of nodes that can be reached from node c via a single action.
  - **Goal Test Function** $G(x)$
    returns true if node c satisfies the goal condition.
  - **Action Cost Function** $C(x, a, y)$
    returns the cost of moving from node x to node y using action a.
    Note that $C(x, a, y) = \infty$ if y is not reachable from a via a.
- **Output**
  - A *sequence of actions* that transforms the initial node satisfying the goal test.
  - The sequence might be, **optimal in cost** for some algorithms, **optimal in length** for some algorithms, come with **no optimality** guarantees from other algorithms.
- **Procedure**
  - Put nodes have not yet expanded in a list called the `Frontier` (or **Open**).
  - Initially, only the *initial node* is in the `Frontier`.
  - At each iteration, pull a node from the `Frontier`, apply $S(x)$, and insert the children back into the `Frontier`.
  - Repeat until pulling a goal node.

Note that the search terminates only when a goal node is expanded into the `Frontier`.

Algorithm 1 Tree Search Algorithm

```
1: function TREE-SEARCH(Frontier, Successors, Goal?)
2:     if Frontier is empty then
3:         return failure
4:     Curr ← select state from frontier
5:     if Goal?(Curr) then
6:         return Curr
7:     Frontier' ← (Frontier − { Curr }) ∪ Successors(Curr)
8:     return TREESEARCH(Frontier', Successors, Goal?)
```

Critical Properties of Search

- **Completeness**
  - A search algorithm is **complete** if it always finds a solution if one exists.
- **Optimality**
  - A search algorithm is **optimal** if it always finds a solution with the lowest cost.
- **Time Complexity**
  - The **time complexity** of a search algorithm is the number of nodes it expands or generates.
- **Space Complexity**
  - The **space complexity** of a search algorithm is the maximum number of nodes it stores in memory.

2.2 Uninformed Search Algorithms

2.2.1 Uninformed Search Strategies

The **order** of nodes in the `Frontier`determines the behaviour of the search algorithm. It determines whether or not the algorithm is complete, optimal, and how much time and space it requires. Search algorithms differ in their selection rule, but they all keep the `Frontier`as an ordered set. The question of which node to select next from the `Frontier`is now equivalent to the question of how to order the `Frontier`.

There are strategies that adpot a *fixed-rule* for selecting the next node to be expanded. These strategies are called **uninformed search strategies** because they do **not** use any *domain specific informatioin* about the problem other than the definition of the problem itself. These strategies include Breadth-First Search, Depth-First Search, Uniform-Cost Search, Depth-Limited Search, and Iterative Deepening Search.

In those course, it is assumed that the graph we are searching is *explicitly* represented as an adjacency list (or adjacency matrix). However, this will **not** work when there an exponential number of nodes and edges. In AI applications, there are typically an exponential number of nodes, making it impossible to explicitely represent them all. AI search algorithms work with *implicitly* defined state spaces, where actions are compacted as *successor state functions*, and nodes must contaion enough information to allow the successor state function to be applied.

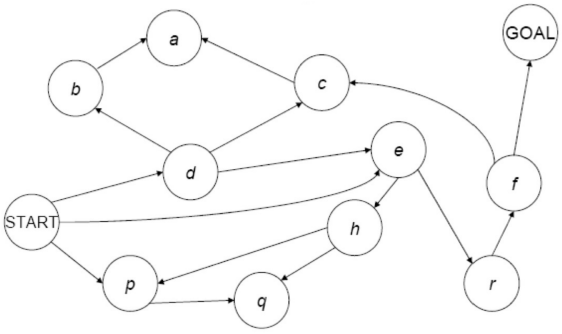
2.2.2 Breadth-First Search (BFS)

Definition 2.2.1 Breadth-First Search

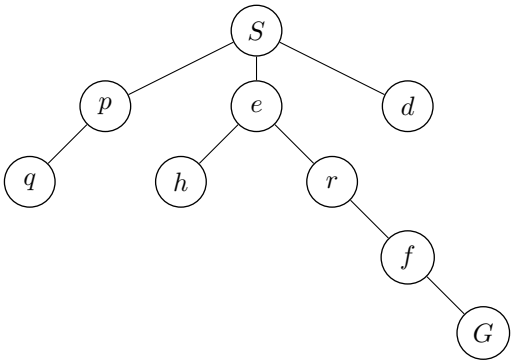
Breadth-First Search is a search strategy that expands the shallowest unexpanded node first.

BFS explores the search tree *level by level*. It place the children of the current node at the **end** of the `Frontier`. This means the `Frontier`is a *queue*, and we always extract the *first* element from the `Frontier`.

Example. We run BFS on the following graph,



which yields the search tree



♢

- **Completeness**
  - BFS is **complete** if the **branching factor** is *finite*.
  - The length of the path (from the initial node to the node removed from the `Frontier`) is *non-decreasing*, as we replace eac expanded node with path langth k with a node with path length of $k+1$.
- **Optimality**
  - BFS gives the **shortest length solution**.
    All nodes with shorter paths are expanded prior to any node with longer path. We examine all paths of length $< k$ before all paths of length k. Thus, if there is a solution with length k, we will find it before longer solutions.
  - BFS is **not optimal** in terms of *cost*.
    BFS does not take into account the cost of the path. It is possible that a longer path has a lower cost than a shorter path.

Properties of BFS

Let b be the maximum branching factor and d be the depth of the shallowest goal node.

- **Time Complexity**

$$1 + b + b^2 + \cdots + b^d + b(b^d - 1) \in O(b^{d+1})$$

The last term id $b(b^d - 1)$ because we need to remove the goal node, who does not need any successors.

- **Space Complexity**

$$\mathcal{O}\left(b^{d+1}\right)$$

In the worst case, only the last node of depth d satisfies the goal. So all nodes at depth d except the last one will be expanded by the search and each such expansion will add up to b new nodes to the `Frontier`.

Depth	Paths	Time	Memory
1	1	0.01 millisec	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

Space complexity is a bigger problem than time complexity. Typically, BFS rus out of memory before it runs out of time.

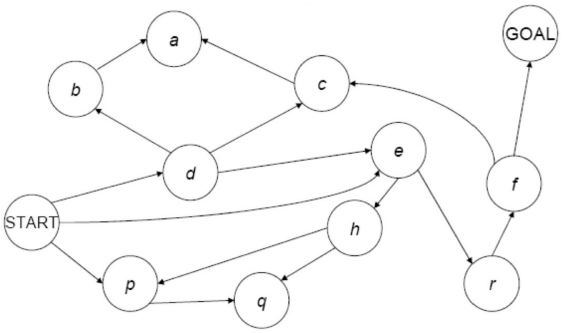
2.2.3 Depth-First Search (DFS)

Definition 2.2.2 Depth-First Search

Depth-First Search is a search strategy that expands the deepest unexpanded node first.

DFS explores the search tree *branch by branch*. It place the children of the current node at the **front** of the `Frontier`. This means the `Frontier`is a **stack**, and we always extract the *first* element from the `Frontier`.

Example. We run DFS on the following graph,



The `Frontier`changes as follows:

Frontier	Node Removed	Successors Added
S	S	d, e, p
d, e, p	d	b, c, e
b, c, e, e, p	b	a
a, c, e, e, p	a	
c, e, e, p	c	a
a, e, e, p	a	
e, e, p	e	h, r

...

♢

- **Completeness**
  - DFS is **not complete**. **Infinite** paths cause incompleteness.
  - This is because DFS does not keep track of visited nodes, so it can get stuck in a loop. An infinite search space can also cause DFS to run forever.
  - We can prune paths with **cycles** to get completeness, if state space is finite.
- **Optimality**
  - DFS is **not optimal** in terms of *length* or *cost*.
  - DFS returns the *first* solution it finds, which may not be the *shortest* or *cheapest* solution.

Properties of DFS

Let b be the maximum branching factor and m be the maximum depth of the search tree.

- **Time Complexity**

  $$\mathcal{O}(b^m)$$

  where m is the length of the *longest path* in the state space.
  - This is **very bad** if m is much larger than d, but if there are *mant solution paths*, DFS may be much faster than BFS.
  - Using **Heuristics**, we can determine which successor to explore first.
- **Space Complexity**

  $$\mathcal{O}(bm)$$

  DFS has a **linear** space complexity, which is a significant advantage.

  This is because DFA only explores a single branch of the search tree at a time. `Frontier`only contains the current node v along with the m descendants of v. Each node can have at most b unexplored siblings and there are at most m nodes on the current branch, which gives $\mathcal{O}(bm)$.

2.2.4 Depth-Limited Search (DLS)

Despite DFS being *incomplete* and *non-optimal*, it has a great advantage over BFS: *linear space complexity*. This makes it possible to use DFS in situations where BFS would run out of memory.

In order to address the incompleteness of DFS, we can impose a *depth limit* D on the search. This means that we will only explore paths of length D or less. If we reach a node at depth D that is not a goal node, we will *backtrack* and explore other paths. This is called **Depth-Limited Search** (DLS).

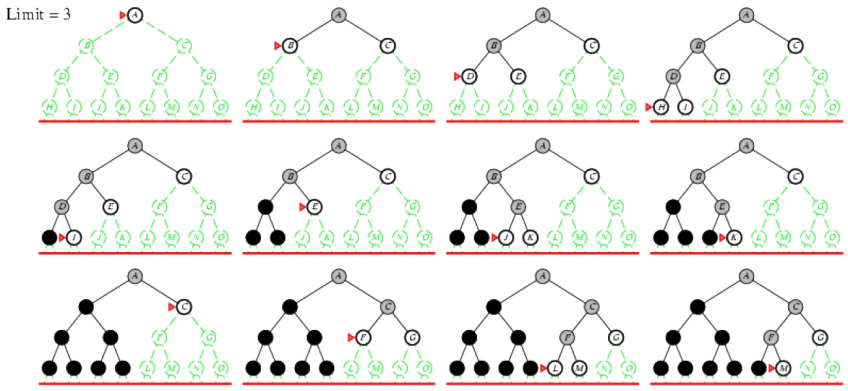
Definition 2.2.3 Depth-Limited Search

Depth-Limited Search is a search strategy that expands the deepest unexpanded node first, but only up to a given depth limit D.

- **Benefit:** DLS overcomes the infinite length paths issue,
- **Limitation:** DLS only finds a solution if the *depth limit* D is greater than or equal to the depth of the shallowest goal node d.

Remark

The root of a search tree is at depth 0. A path consisting onlty of the root node has length 0. Recall that the length of a path is the number of actions (edges) in the path.



Algorithm 2 Depth-Limited Search Algorithm

```
1:  function DLS(start, frontier, successors, goal?, maxd)
2:      frontier.insert(<start>)                          ▷ frontier must be a stack for DFS
3:      cutoff ← false
4:      while frontier is not empty do
5:          curr ← frontier.extract()                     ▷ Remove node from frontier
6:          if goal?(curr.final()) then
7:              return curr, cutoff                       ▷ curr is solution
8:          else if LENGTH(curr) < maxd then              ▷ Only successors if lendth(curr) < maxd
9:              for succ ∈ successors(curr.final()) do
10:                 frontier.insert(<p, succ>)
11:         else
12:             cutoff ← true                             ▷ Reached depth limit, some nodes not expanded
13:     return failure, cutoff
```

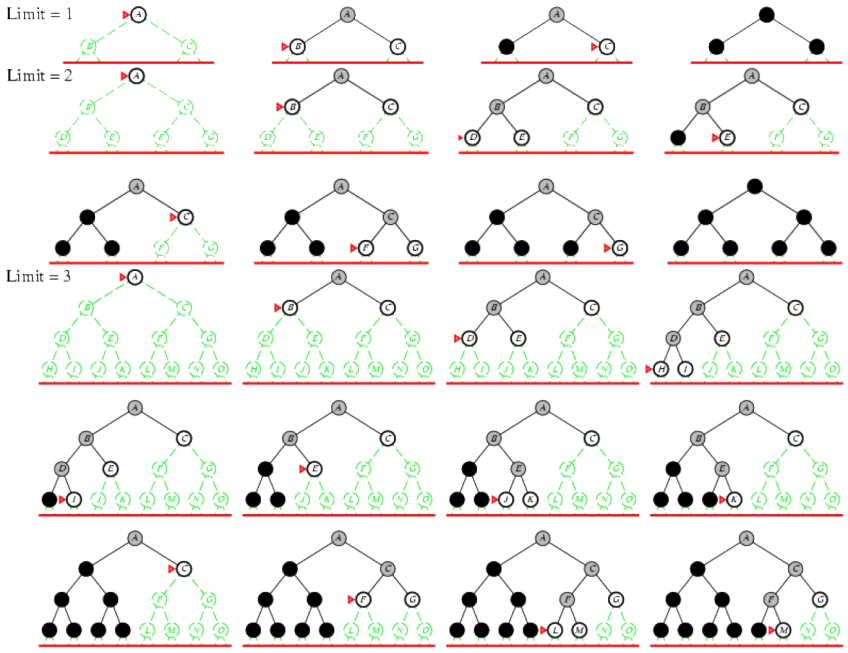
2.2.5 Iterative Deepening Search (IDS)

DLS is *complete* if the *depth limit* D is greater than or equal to the depth of the shallowest goal node d. However, we do not know d in advance. We can start at a depth limit $D = 0$, and iteratively **increase** D until we find a solution. We stop if **a solution is found**, or the search **failed without cutting off** any nodes because of the depth limit. This is called **Iterative Deepening Search** (IDS).

Note that if no nodes were cut off, the search examined all nodes in the state space and found nosolution, hence there is no solution.

Definition 2.2.4 Iterative Deepening Search

Iterative Deepening Search is a search strategy that repeatedly applies DLS with increasing depth limits until a solution is found.



Algorithm 3 Iterative Deepening Search Algorithm

```
function IDS(start, frontier, successors, goal?)
    depth ← 0
    while true do
        solution, cutoff ← DLS(start, successors, goal?, depth)
        if solution is not failure then
            return solution
        else if cutoff is false then                ▷ No nodes at deeper levels exist, no solution
            return failure
        depth ← depth + 1
```

- **Completeness**
  - IDS is **complete**.
- **Optimality**
  - IDS gives the **shortest length solution**.
  - IDS is **not optimal** in terms of *cost*, for the same reason as DFS.
    We can make IDS optimal by doing the following:
    - Can use a **cost bound** instead of depth bound.
    - Only expand nodes of cost *less* than the cost bound.
    - Keep track of the **minimum cost unexpanded node** in each iteration, *increase the cost bound* to that on the next iteration.
    - Can be **computationally expensive** since need as many iterations of the search as there are distinct node costs.

Properties of IDS

- **Time Complexity**

$$(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d \in \mathcal{O}\left(b^d\right)$$

  - This is **very bad** if d is much larger than m, but if there are *mant solution paths*, IDS may be much faster than BFS.
  - Using **Heuristics**, we can determine which successor to explore first.

- **Space Complexity**

$$\mathcal{O}\left(bd\right)$$

  The space complexity of IDS is still linear.

> **Remark**
> Comparing to BFS,
>
> - The time complexity of IDS can be better since it does not expand nodes at the solution depth, while BFS (in the worst case) must expand all the *bottom later nodes* until it expands a goal node. However, with a simple optimization BFS can achieve the *same time complexity* as IDS.
> - The space complexity of IDS is much better.
>   - In practice, BFS can be buch better depending on the problem: effective **cycle checking** can be employed with BFS
>   - IDS cycle checking will make the space complexity as bad as BFS.

2.2.6 Path Checking and Cycle Checking

Path Checking

> **Definition 2.2.5 Path Checking**
> **Path Checking** is a search strategy that checks whether a node is an ancestor of itself along the path from the initial node to the node.

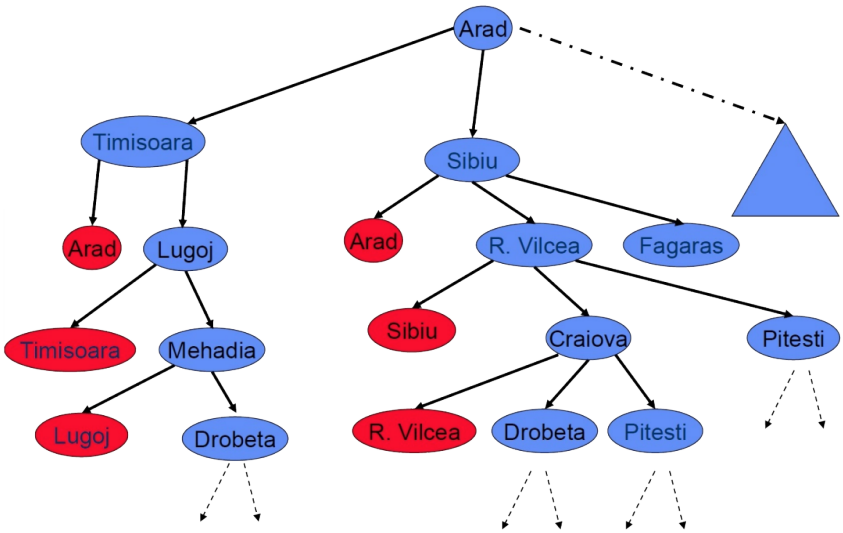
> **Remark Notation**
>
> - A path p_k is represented as a typle of states $\langle s_0, s_1, \cdots, s_k\rangle$ where $s_0, s_1, \cdots, s_k$ are states in p_k in the order they appear in the path.
> - Suppose s_k is expanded to obtain a child succcessor statce c. The obtained path $\langle s_0, s_1, \cdots, s_k, c\rangle$ can be written as $\langle p_k, c\rangle$.

- In every path $\langle p_k, c\rangle$ where p_k is the path $\langle s_0, s_1, \cdots, s_k\rangle$, ensure that the **final state** c is not equal to any ancestors of c along this path. That is,

$$c \notin \{s_0, s_1, \cdots, s_k\}$$

- Paths are checked in **isolation**
- **Benefit**: Does not increase time and space commplexity.
- **Limitation**: Does not prune all the redundant states.

Example. Consider the Romania Travel problem. Below is an example of a search tree.



The nodes coloured in red are pruned by path checking, as they are ancestors of the current node. ♢

Cycle Checking

Cycle checking is also called **Multiple path checking**. It is a *more general* version of path checking. It checks whether a node has been expanded before, not just whether it is an ancestor of itself along the path from the initial node to the node.

Definition 2.2.6 Cycle Checking

Cycle Checking is a search strategy that checks whether a node has been expanded before.

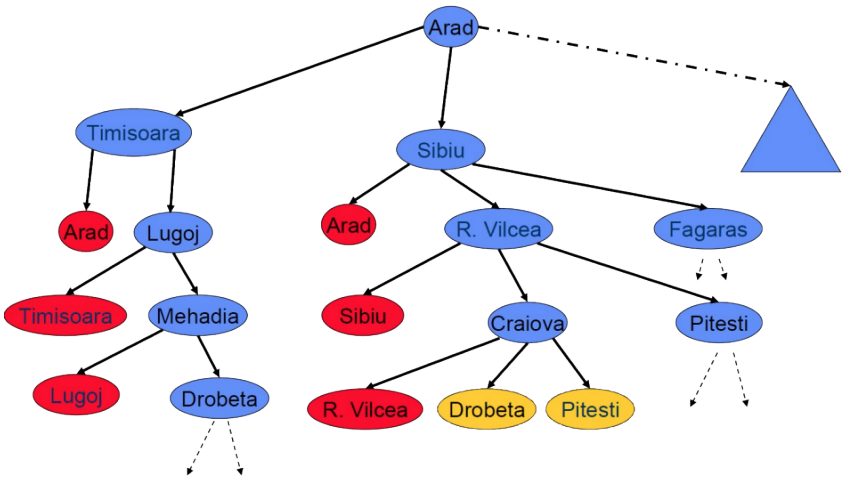
- Keep track of **all** nodes previous expanded during the search using a list called the **closed list**.
- When we expand n_k to obtain successor c,
  - Ensure that c is not equal to any *previously expanded* node.
  - If it is, we **do not add** c to the `Frontier`.
- **Benefit**: Very *effective* in pruning redundant states.
- **Limitation**: Expensive in term of **space**.

  The space complexity is $\mathcal{O}\left(b^d\right)$ with optimization, and $\mathcal{O}\left(b^{d+1}\right)$ without optimization (which is the same as BFS).

Remark

We cannot utilize this technique with DFS. This is because DFS only puts a node in the closed list after all its descendants have been expanded. Thus, the "root" node will be put on the list later than its descendants, and cycle checking will fail.
Moreover, the space complexity of DFS with cycle checking would be exponential, which will loose the advantage of DFS.

Example. Consider the Romania Travel problem. Below is an example of a search tree.



The nodes coloured in red are the explored ancestors, and the nodes coloured yellow are the nodes already exploare in another branch. ♢

Optimal Cost

If we reject a path $\langle p, c\rangle$ because we have previously seen state c via a different path p', it could be that $\langle p, c\rangle$ is a cheaper path to c than p

The solution would be keeping track of each state as well as the **known minimum cost** of a path to that state.

- If a more **expensive** path to a previously seen state is found, do not add the corresponding node to the `Frontier`.
- If a cheaper path to a previously seen state is found, add the corresponding node to the `Frontier`and
  - *Remove* other more *expensive* nodes to the same state from the `Frontier`, OR
  - Lazily, *ignore* these more expensive nodes when/if they are removed for expansion.

2.2.7 Uniform-Cost Search (UCS)

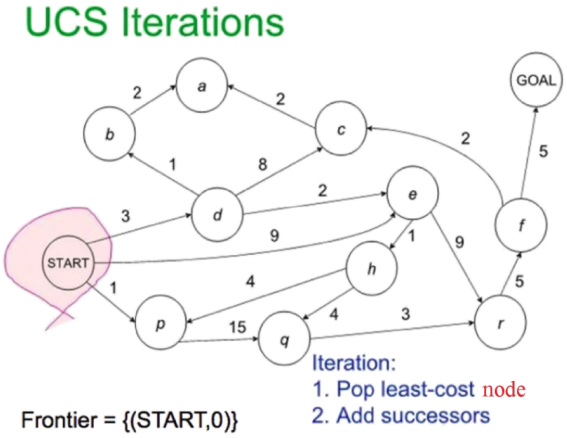
The **Uniform-Cost Search** (UCS) algorithm is a variant of BFS that uses *cost* as the criterion for selecting the next node to expand. UCS expands the node with the *lowest path cost* $g(n)$, where $g(n)$ is the cost of the path from the initial node to node n.

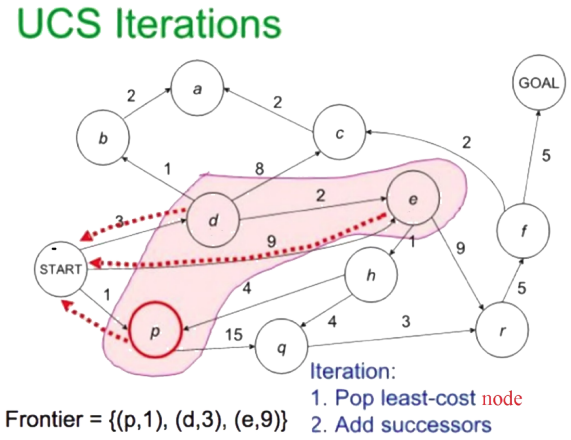
Definition 2.2.7 Uniform-Cost Search

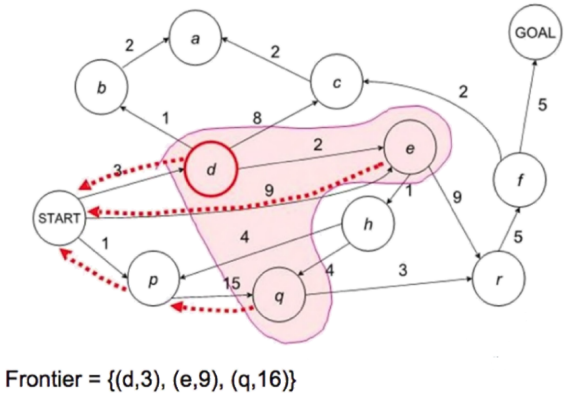
Uniform-Cost Search is a search strategy that expands the node with the lowest path cost $g(n)$.

Remark

UCS is a *generalization* of BFS. BFS is UCS with a *constant cost* of 1 for each action.







...

- **Completeness**
  - UCS is complete under the condition that there is a **non-zero constant lower-bound** ϵ on the cost of every action.
    That is, each transition from one state to another must have a cost of at least $\epsilon > 0$.
  - Under this condition the cost of the nodes chosen to be expanded will be non-decreasing and eventually we will expand all nodes with cost equal to the cost of a solution path
- **Optimality**
  - UCS finds the **minimum cost** solution if each transition has cost $\geq \epsilon > 0$.
  - UCS explores nodes in the search space in non-decreasing order of cost. So it must find minimum cost path to a goal before finding any higher costs paths leading to a goal.

Proof. UCS is optimal if the cost of each action is $\geq \epsilon > 0$.

> **Remark** **Intuition**
>
> - nodes are expanded in order of **non-decreasing cost**.
> - **All nodes** with cost strictly less than C are expanded before **all nodes** with cost equal to C.
> - The first path found to a state is the **cheapest** path to that state (be it a goal state or not).

In all of the following lemmas the assumption is that each transition has cost $\geq \epsilon > 0$.

Lemma 2.2.1

Let $c(n)$ denote the cost of a node n on the `Frontier`. If a node n_2 is expanded after n_1 by UCS, then $c(n_1) \leq c(n_2)$.

Lemma 2.2.2

Let n be an arbitrart node expanded by UCS in a search tree. All the nodes in the search space with cost strictly less than $c(n)$ are expanded **before** n.

Lemma 2.2.3

Let p be the first path UCS find whose final state is a state s. Then p is a **minimal cost** path to s.

■

Properties of UCS

The time and space complexity of UCS are both

$$\mathcal{O}\left(b^{\lfloor C^*/\epsilon \rfloor + 1}\right)$$

- UCS has to expand *all nodes* with cost **less** than C^* and potentially *all nodes* with cost **equal** to C^*.
- In the worst case, there are $\mathcal{O}\left(b^{\lfloor C^*/\epsilon \rfloor + 1}\right)$ nodes with cost *less than or equal to* C^*.

Remark

In the worst case, there are $\lfloor C^*/\epsilon \rfloor$ actions on the optimal path, and each state has b successors, where C^* denotes the cost of the optimal solution path. We also assume each action has cost $\geq \epsilon > 0$.

2.3 Heuristic Search

2.3.1 Heuristic Search Strategies

Uniform search methods never *look ahead* to the goal. They enever try to evaluate which nodes on the `Frontier`are the most promising. Heuristic search methods, on the other hand, use *domain-specific knowledge* to evaluate which nodes are the most promising.

- If we are concerned about the **cost** of the solution, we might want to consider how costly it's to extend a node so that it reaches a goal node.

  That is, the cost of getting from the node to a goal.

- If we are concerned about **minimizing computation** in search we might want to consider how easy it is to extend that node so that it reaches a goal.

  That is, how hard is it to get from the node to a goal.

The idea of a heuristic search is to develop a *domain-specific* **heuristic function** $h(n)$ that estimates the cost of the cheapest path from node n to a goal node. $h(n_1) < h(n_2)$ means that we **guess** n_1 is *closer* to a goal than n_2.

Definition 2.3.1 Heuristic Function

A **heuristic function** $h(n)$ is a function that estimates the cost of the cheapest path from node n to a goal node.

Remark

$h(n)$ is a function **only** of the state of n. That is, if the states of n_1 and n_2 are the same, then $h(n_1) = h(n_2)$.

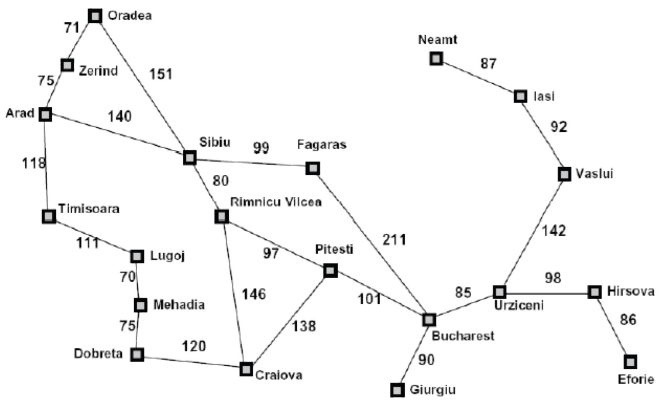
Sometimes, we might even use a *state*, rather than a node, as the argument of the heuristic function.

Remark

h **must** be defined so that $h(n) = 0$ for any goal node n.

There are different ways of guessing heuristic cost in different domains. For example, Alpha Go exploited machine learning techniques to compute the heuristic estimate of a state (go board configuration).

Example. Consider the Romania Travel problem. The heuristic function $h(n)$ is defined to be the Euclidean distance between the current city and Bucharest.



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

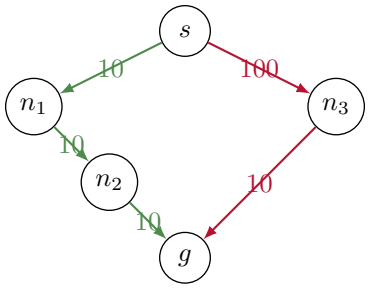
2.3.2 Greedy Best-First Search (GBFS)

Greedy Best-First Search (GBFS) uses $h(n)$ to rank the nodes on the `Frontier`. It always expands the node with the *lowest heuristic cost* $h(n)$.

Definition 2.3.2 Greedy Best-First Search

Greedy Best-First Search is a search strategy that expands the node with the lowest heuristic cost $h(n)$.

GBFS greedily tries to find the *cheapest* path to a goal. It *ignores* the cost of n, so it can be **lead astray** by paths that look promising based on $h(n)$ but actually have a high cost.



Greedy search can be very efficient in practice at finding solutions, but it requires developing a good heuristic function. Greedy search is *not complete* and *not optimal*. The solution returned by a greedy search can be **very far from optimal**.

Example. GBFS will be stuck in the following graph, where the optimal heuristic leads to an infinite loop, while only the suboptimal heuristic leads to the goal.

2.3.3 A* Search

A* Search is a variant of UCS that uses $h(n)$ to rank the nodes on the `Frontier`. It always expands the node with the *lowest evaluation cost*

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

where $g(n)$ is the cost of the path from the initial node to node n. Here, $f(n)$ is an estimate of the cost of getting to the goal through n.

Definition 2.3.3 A* Search

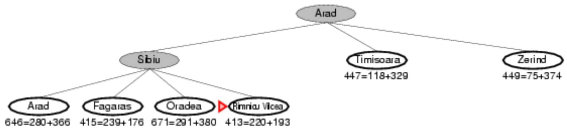
A* Search is a search strategy that expands the node with the lowest evaluation cost

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

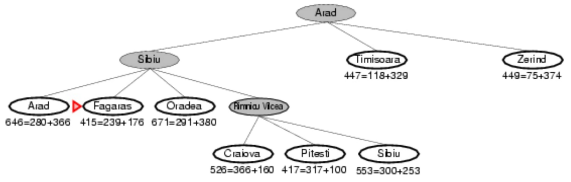
Example. Consider a tracing example of Romania Travel problem. The heuristic function $h(n)$ is defined to be the Euclidean distance between the current city and Bucharest.

Arad

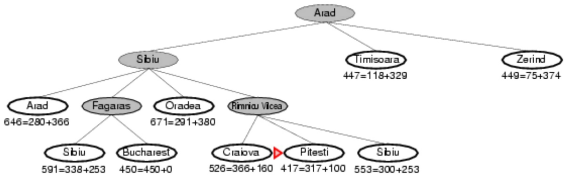
Sibiu



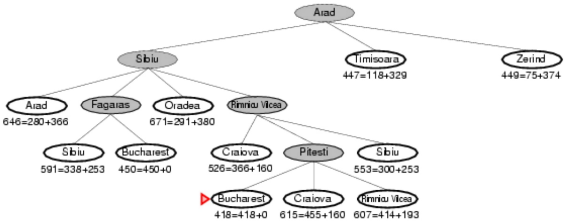
Rimriau Vâlcea



Făgăraș



Pitești



Bucharest

Remark **No Solution Case**

A*, as well as all of the uninformed search algorithms, have the same behaviour when there is **no solution**:

- If there are an *infinite* number of different *states* reachable from the initial state, then these algorithms **never terminate**.
- If there are a *finite* number of different reachable states (and nodes) and we do either *path checking* or *cycle checking*, they will eventually terminate and correctly declare that there is no solution (assuming costs are always $\geq \epsilon > 0$).

- **Completeness**

Theorem 2.3.1 **Completeness of A* Search**

A* will always find a solution if one exists as long as

- the branching factor is *finite*;
- every action has a *finite cost* greater than $\epsilon > 0$; and
- $h(n)$ is *finite* for every node n that can be extended to **reach a goal node**.

Proof. If a solution n exists, then at all times either

- n has been expanded by A*, or
- an ancestor of n is on the `Frontier`.

Suppose the latter holds, and let the ancestor on the `Frontier` be n_i.

Then, n_i must have a finite f-value.

As A* continues to run, the f-value of the nodes on the `Frontier` eventually **increase**. So, eventually, either A* terminates because it found a solution, or n_i becomes the node **on the `Frontier` with the lowest f-value**.

If n_i is expanded, then either $n_i - n$ and A* returns n as a solution, or n_i is replaced by its successors, one of which n_{i+1} is a *closer ancestor* of n.

Applying the same argument to n_{i+1}, we see that if A* continues to run without finding a solution, it will **eventually expand every ancestor of** n, including n itself.

Therefore, A* will eventually return a solution if one exists. ■

- **Optimality**

Definition 2.3.4 **Asmissibility**

Let $h^*(n)$ be the cost of an optimal path from n to a goal node (∞ if there is no path). An **admissible** heuristic is a heuristic that satisfies the following condition for all nodes n in the search space:

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

In other words, an admissible heuristic is a heuristic that **never overestimates** the cost of reaching a goal node from n.

To achieve optimality, we must pou come conditions on $h(n)$ and the search space.

- Each action in the search space must have a *cost* $\geq \epsilon > 0$.

- h must be *admissible*.

This way, the search will not miss any promising paths. If it really is cheap to get to a goal via n (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n)$ is also low, and eventually n will be expanded.

Example. Consider when there are two goal nodes, g_1 and g_2, with $\textsc{Cost}(g_1) < \textsc{Cost}(g_2)$. Let n be a node on the path to g_1.

Then, by having an admissible heuristic, we will always expand n before expanding any node on the path to g_2 (even if g_2 itself is on the `Frontier`). ♢

Proposition 2.3.1

A* with an admissible heuristic never expands a node with f-value greater than the cost of an optimal solution.

Proof. Let C^* be the cost of an optimal solution.

Let $p : \langle s_0, s_1, \cdots, s_k \rangle$ be an optimal solution path. Then, $\textsc{cost}(p) = \textsc{cost}(\langle s_0, s_1, \ldots, s_k \rangle) = C^*$.

It can be shown bt induction for each node in the search space that is reachable from the initial node, at every iteration an ancestor of the node is on the `Frontier`.

Let n be a node reachable from the initial state and $n_0, n_1, \ldots, n_i, \ldots, n$ be ancestors of n. SO at least one of $n_0, n_1, \ldots, n_i, \ldots, n$ is **always** on the `Frontier`.

We show that with an admissible heuristic, for every ancestor n_i of n on the `Frontier`, $f(n_i) \leq C^*$

$$\begin{aligned} C^8 &= \textsc{Cost}\left(\langle s_0, s_1, \ldots, s_k \rangle\right) \\ &= \textsc{Cost}\left(\langle s_0, s_1, \ldots, s_i \rangle\right) + \textsc{Cost}\left(\langle s_{i+1}, s_{i+2}, \ldots, s_k \rangle\right) \\ &= g(n_i) + h^*(n_1) \\ &\geq g(n_i) + h(n_i) = f(n_i) \end{aligned}$$

We know that A* always expands a node on teh `Frontier` that has *lowest f-value*. So every node A* expands has f-value $\leq f(n_i) \leq C^*$. ■

Theorem 2.3.2 Optimality of A* Search

A* with an admissible heuristic always finds an optimal cost solution if one exists, as long as

- the branching factor is *finite*; and
- every action has a *finite cost* greater than $\epsilon > 0$

Proof. Let C^* be the cost of an optimal solution.

If a solution exists, then by completeness, A* will eventually expand some solution node n.

By the proposition, $f(n) \leq C^*$. Since n is a goal node, we have $h(n) = 0$. So $f(n) = g(n) = \textsc{Cost}(n)$. Thus, $\textsc{Cost}(n) \leq C^*$.

We also have that $C^* \leq \textsc{Cost}(n) = f(n)$ since no solution can have lower cost than the optimal solution.

Thus, $\textsc{Cost}(n) = C^*$, and n is an optimal solution.

That is, A* returns a cost-optimal solution. ■

Definition 2.3.5 Monotonicity / Consistency

A **monotone** (or **consistent**) heuristic h is a heuristic that satisfies the triangle inequality: for all nodes n_1, n_2 and all actions a we have that

$$h(n_1) \leq C(n_1, a, n_2) + h(n_2)$$

where $C(n_1, a, n_2)$ is the cost of getting from the state n_1 to the state n_2 by applying action a.

In other words, a monotone heuristic will not suddenly decrement the estimated cost by taking a single action. A consistent heuristic will not be misleading, thus improves the efficiency of A*.

Theorem 2.3.3

Monotonicity implies admissibility. That is,

$$(\forall n_1, n_2 a) h(n_1) \leq C(n_1, a, n_2) + h(n_2) \implies (\forall n) h(n) \leq h^*(n)$$

If a node n_2 is expanded after n_1 by $A*$ with a monotonic heuristic, then $f(n_1) \leq f(n_2)$. With a monotonic heuristic, the first time A* expands a node n, n must be a *minimum cost* solution to $n.state$.

Properties of A* Search

- **Completeness and Optimality**

  Yes, as shown in the previous theorems.

- **Time Complexity and Space Complexity**

  When $h(n) = 0$ for all nodes n, A* is equivalent to UCS.

  It can be shown that when $h(n) > 0$ (for some n) and an admissible heuristic is used, the number of nodes expanded can be no longer than UCS, hence the same bounds on time and space complexity apply. We still have an **exponential** time and space complexity, unless we have a *good heuristic*. In real-world problems, we sometimes run out of time and memory before we find a solution.

2.3.4 Iterative Deepening A* (IDA*)

The objective of **Iterative Deepening A*** (IDA*) is to *avoid* the *exponential* space complexity of A* by using *iterative deepening*. Similar to iterative deepening search, IDA* repeatedly applies A* with increasing f-value limits until a solution is found. At each iteration, the cutoff value is the *smallest f-value* of any node that exceeded the cutoff in the *previous iteration*. This will avoid overhead associated with keeping a sorted queue of nodes, and the `Frontier` will occupy only **linear space**.

Definition 2.3.6 Iterative Deepening A*

Iterative Deepening A* (IDA*) is a search strategy that repeatedly applies A* with increasing f-value limits until a solution is found.

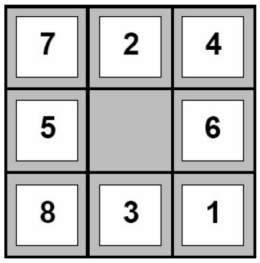
2.3.5 Building Heuristic Functions

Remark **How to Build A Heuristic**

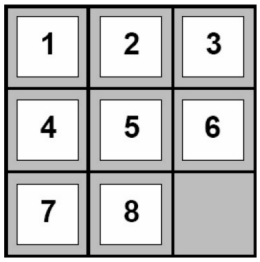
Simplify a problem when building heuristics and let $h(n)$ be the cost of reaching the goal in the easier problem.

Example. In the 8-Puzzle we can only move a tile from square A to B if A is adjacent (left, right, above, below) to B and B is blank. We can relax some of these conditions and:

1. allow a move from A to B if A is adjacent to B (ignore whether or not position is blank), which leads to the Manhattan distance heuristic
2. allow a move from A to B if B is blank (ignore adjacency);
3. allow all moves from A to B (ignore both conditions), which leads to the misplaced tiles heuristic



Start State



Goal State

Admissible Heuristics for the 8-puzzle Problem include:

- $h(n)$: number of misplaced tiles in $n.state$ (misplaced tiles)
- $h(n)$: total Manhattan distance of all tiles from their goal positions in $n.state$ (Manhattan distance)

CHAPTER

3 GAME TREE SEARCH

CHAPTER

CONSTRAINT SATISFACTION PROBLEMS

4

CHAPTER

REPRESENTING AND REASONING UNDER UNCERTAINTY

5

CHAPTER

SYMBOLIC KNOWLEDGE REPRESENTATION AND REASONING

6

Part II

Tutorials

Tutorial 1

Search Problem

In order to formalize a problem as a search problem, we need to define the following:

1. **State Space**: A state is a representation of a configuration of the problem domain. The state space is the set of all states included in our model of the problem.
2. **Initial State**: The starting configuration.
3. **Goal State**: The configuration one wants to achieve.
4. **Actions** (or **State Space Transitions**): Allowed changes to move from one state to another.

Optionally, we may use the following:

1. **Costs**: Representing the cost of moving from state to state.
2. **Heuristics**: Help guide the heuristic process

Example (Make Change)**.** We start with 0 cents and we want to reach some number of cents between $[0, 5, 10, 15, ..., 500]$ using the least number of coins. We can add $5, 100, 25, 100$, or 200 cents.
Search problem formulation with the goal of X cents:

- States: Integers between 0 and 500 that are divisible by 5.
- Initial State: 0
- Goal State: X
- Actions: Add 5, 10, 25, 100, or 200

Questions

1. Sudoku is a popular number puzzle that works as follows: we are given a 9×9 square grid; some squares have numbers, while some are blank. Our objective is to fill in the blanks with numbers from $1 - 9$ such that each row, column, and the highlighted 3×3 squares contain no duplicate entries (see Figure 1).

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

Figure 1: A simple Sudoku puzzle.

2	5	8	7	3	6	9	4	1
6	1	9	8	2	4	3	5	7
4	3	7	9	1	5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

Figure 2: Solution to the puzzle in Figure 1.

Sudoku puzzles can be solved easily after being modelled as a CSP (which will be covered later in the course). We will consider the problem of generating Sudoku puzzles. In particular, consider the following procedure: start with a completely full number grid (see Figure 2), and iteratively make some squares blank. We continue blanking out squares as long as the resulting puzzle can be completed in at least one way.

Complete the following:

- Give the representation of a state in this problem.
  Solution: A state in this problem is a (partial) valid solution of the Sudoku puzzle. More formally, it would be a matrix $M \in \{0, \ldots, 9\}^{9\times 9}$, with 0 representing a blank square.
- Using the state representation defined above, specify the initial state and goal state(s).
  Solution:
  - The initial state is a completed filled out grid of numbers which is a valid. All rows, columns, and 3×3 squares should contain all numbers between $1, \ldots, 9$.
  - Any state in the problem will be a valid goal state.
- Define its actions.
  Solution: An action would be removing a number from the grid. More formally, we take as input a matrix M and a matrix $E_{i,j}(a)$, where $E_{i,j}(a) \in \{0, \ldots, 9\}^{9\times 9}$ is a matrix of all zeros except for a non-zero value $a \in \{1, \ldots, 9\}$ in coordinate (i, j). An action would be setting $M - E_{i,j}(a)$.
- Using the state representation and actions defined above, specify the transition function T. (In other words, when each of the actions defined above is applied to a current state, what is the resulting state?)
  Solution: The transition model would be $T(M - E_{ij}(a)) = M - E_{ij}(a)$.

2. Assuming that ties (when pushing to the frontier) are broken based on alphabetical order, specify the order of the nodes that would be explored by the following algorithms. Assume that S is the initial node, while G is the goal node. You should express your answer in the form $S - B - A - F - G$ (i.e. no spaces, all uppercase letters, delimited by the dash $(-)$ character), which, for example, corresponds to the exploration order of S, B, A, F, then G.

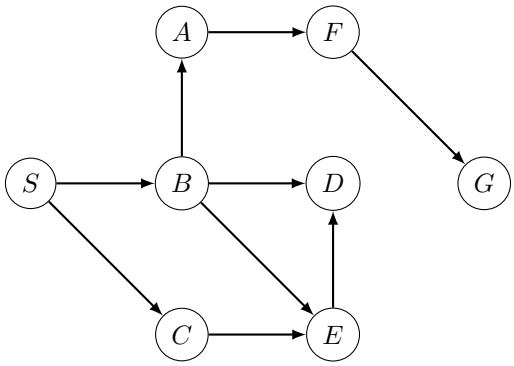


Figure 3: Graph for question 2.

(a) Depth-First Search with no cycle-checking (aka tree-based) implementation.
Solution: $S - C - E - D - B - E - D - A - F - G$

(b) Breadth-First Search with no cycle-checking (aka tree-based) implementation.
Solution: $S - B - C - A - D - E - F - G$

(c) Breadth-First Search with cycle-checking (aka graph-based) implementation.
Solution: $S - B - C - A - D - E - F - G$

3. (a) The **Breadth-First Search** algorithm is complete if the state space has infinite depth but finite branching factor.
Determine if the statement above is True or False, and provide a rationale.
Solution: This is **true**.
This follows immediately from the definitions of Breadth-First Search and completeness. (Recall that a search algorithm is complete if whenever there is a path from the initial state to the goal, the algorithm will find it.)

In particular, the existence of a path means that the goal exists in a finite depth, and therefore Breadth-First Search will eventually reach it, since having a finite branching factor implies that the algorithm would not get lost in infinite-breadth search at some level.

(b) The **Breadth-First Search** algorithm can be complete even if zero step costs are allowed.
Determine if the statement above is True or False, and provide a rationale

Solution: This is **true**.

As highlighted in the previous question, Breadth-First Search is complete as long as the state space has a finite branching factor. Note that Breadth-First Search (as well as Depth-First Search) does not take into account edge weights.

(c) Given that a goal exists within a finite search space, the **Breadth-First Search** algorithm is cost-optimal if all step costs from the initial state are non-decreasing in the depth of the search tree. That is, for any given level of the search tree, all step costs are greater than the step costs in the previous level.
Determine if the statement above is True or False, and provide a rationale

Solution: This is **false**.

We could have two goal nodes in the same level with different costs but with the same depth. For example, arbitrary tie-breaking (or breaking ties by alphabetical ordering) of the Breadth-First Search algorithm may result in A reached before B.

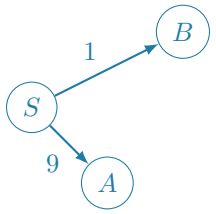


Figure 4: A counter-example for question 3c.

4. Prove that the **Uniform Cost Search** algorithm is cost-optimal as long as each step cost exceeds some small positive constant ϵ.

Solution:

Proof. WTS UCS is optimal as long as each step cost exceeds positive value ϵ.

Given that each step cost exceeds some small positive constant ϵ, completeness may be assumed. Consequently:

- Whenever UCS expands a node n, the optimal path to that node has been found. If this was not the case, there would have to be another frontier node n' on the optimal path from the start node to n. By definition, n' would have a lower value of g than n, and thus would have been selected first.
- If step costs are non-negative, paths never get shorter as nodes are added.

The above two points imply that UCS expands nodes in the order of their optimal path cost. ■

Tutorial 2

UCS

Consider the search problem of Figure 1. Execute UCS (without cycle-checking) starting from node S, and the goal node being G.

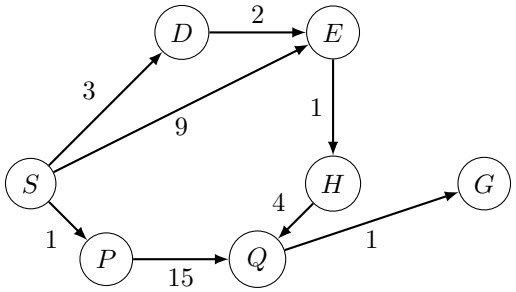


Figure 1: A search problem.

Solution:

(Node, Path)	Frontier
	$\{(S,0)\}$
(S,S)	$\{(SP,1),(SD,3),(SE,9)\}$
(P,SP)	$\{(SD,3),(SE,9),(SPQ,16)\}$
(D,SD)	$\{(SDE,5),(SE,9),(SPQ,16)\}$
(E,SDE)	$\{(SDEH,6),(SE,9),(SPQ,16)\}$
$(H,SDEH)$	$\{(SE,9),(SDEHQ,10),(SPQ,16)\}$
(E,SE)	$\{(SEH,10),(SDEHQ,10),(SPQ,16)\}$
(H,SEH)	$\{(SDEHQ,10),(SEHQ,14),(SPQ,16)\}$
$(Q,SDEHQ)$	$\{(DEHQG,11),(SEHQ,14),(SPQ,16)\}$
$(G,SDEHQG)$	$\{(SEHQ,14),(SPQ,16)\}$

A* & Heuristics

A tracing example of executing A* (without cycle-checking) on the problem of figure 2

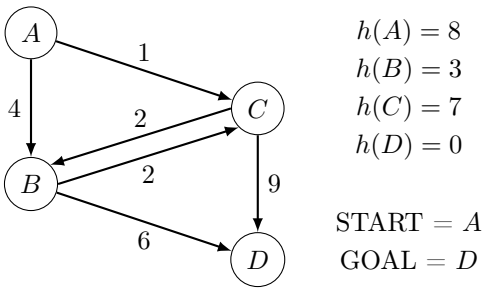


Figure 2: Another search problem.

Each cost is depicted as the cost to get that node plus the heuristic cost at that node. Next node to be expanded is highlighted in blue. Here we maintain (Node, Path, Cost) as the tuple in the frontier.

Node Popped	Frontier
	$\{(A, A, 0+8=8)\}$
A	$\{(C, AC, 1+7=8), (B, AB, 4+3=7)\}$
B	$\{(C, AC, 1+7=8), (C, ABC, 6+7=13), (D, ABD, 10+0=10)\}$
C	$\{(B, ACB, 3+3=6), (D, ACD, 10+0=10), (C, ABC, 6+7=13), (D, ABD, 10+0=10)\}$
B	$\{(C, ACBC, 5+7=12), (D, ACBD, 9+0=9), (D, ACD, 10+0=10), (C, ABC, 6+7=13), (D, ABD, 10+0=10)\}$

1. Execute A* with cycle-checking on the problem of figure 2.

   Solution:

Node Popped	Frontier
	$\{(A, A, 0+8=8)\}$
A	$\{(C, AC, 1+7=8), (B, AB, 4+3=7)\}$
B	$\{(C, AC, 1+7=8), (D, ABD, 10+0=10)\}$
C	$\{(B, ACB, 3+3=6), (D, ABD, 10+0=10)\}$
B	$\{(C, ACBC, 5+7=12), (D, ACBD, 9+0=9)\}$

2. Prove that the A* Search algorithm with cycle-checking is optimal when a consistent heuristic is utilized.

3. In the search problem below, we have listed 5 heuristics. Indicate whether each heuristic is admissible and/or consistent in the table below.

   Remark

   - Consistent: $h(v_i) \leq h(v_j) + c(v_i, v_j)$
   - Admissible: $h(v) \leq c^*(v)$

	S	A	B	G	Admissible	Consistent
h_1	0	0	0	0	True	True
h_2	8	1	1	0	True	False
h_3	9	3	2	0	True	False
h_4	6	3	1	0	True	True
h_5	8	4	2	0	False	False

4. Given that a heuristic h is such that $h(s_g) = 0$ where s_g is any goal state, prove that if h is consistent, then it must be admissible

5. a) Provide a counter-example to show that the **Greedy Best-First Search** algorithm without cycle-checking is incomplete.

   Solution:

   b) Provide a counter-example to show that Greedy Best-First Search (with or without cycle-checking) is not optimal.

Part III

Appendices

BIBLIOGRAPHY

[TUR50] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE". In: *Mind* LIX.236 (Oct. 1950), pages 433–460. ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf. URL: https://doi.org/10.1093/mind/LIX.236.433 (cited on page 7).

INDEX

A
A* Search, 27
Asmissibility, 28

B
Breadth-First Search, 16

C
Consistency, 30
Cycle Checking, 22

D
Depth-First Search, 17
Depth-Limited Search, 19

G
Greedy Best-First Search, 26

H
Heuristic Function, 26

I
Iterative Deepening A*, 30
Iterative Deepening Search, 20

M
Monotonicity, 30

P
Path Checking, 21

S
Search Tree, 13

U
Uniform-Cost Search, 23