ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

State Space Search

Prof. Dr. Goran Glavaš

7.12.2023

Content

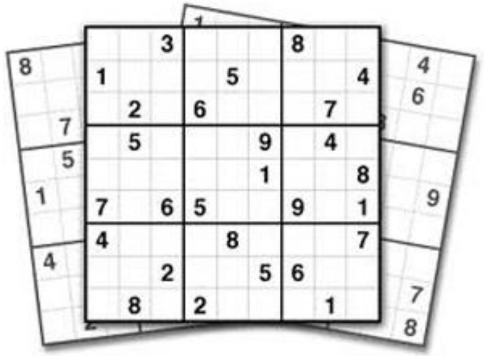
- State Space Search: Fundamental AI Problem
- Uninformed Search
- Example: missionaries & cannibals

Based on the materials from Prof. Dr. Jan Šnajder:
https://www.fer.unizg.hr/_download/repository/AI-2-StateSpaceSearch.pdf

Motivation

- Many analytical ("AI") problems can be solved by **searching** through a **space of possible states**
- Starting from an **initial state**, we try to reach a **goal state**
- Sequence of actions leading from initial to goal state is the **solution** to the problem
- Problem: **large number of states** and **many choices to make** in each state
- Search must be carried out in a **systematic manner**

Typical state space search problems



State Space Search vs. Divide-and-Conquer

State space search

- **Many different** possible **”states”** in which „solving” of the problem can be
  - E.g., Imagine all different states in which a chessboard can be?
- **Many different transitions** from each state (to many possible next states)
  - Large **„branching factor”**
- Finding a node that **meets a particular criterion** (goal node)
  - Some kind of optimality can be an additional criterion
  - But iterating through all states not feasible

Divide-and-Conquer

- **Break the problem** into subproblems of the **same type**
- „state” = **subproblem** to solve
  - And its relation to the global problem
- Typically, a small(er) branching factor
  - Problem broken into a small number of subproblems
- Typically **needs to visit all states** (solve all subproblems)
  - If „states” are revisited (**repeating subproblems**) –> dynamic programming
- Not searching for a special (goal) state but finding an **optimum** over all states

State Space Search: Formalization

- We will denote the set of all states (state space) with **S**
  - The state space is commonly **so large** that we can't iteratively list all states
  - All states in the space are not really „known" in advance
  - When in state s, we typically only then compute the set of possible next states

State space search

A **state space search problem** is defined with a triple (s_0, *succ*, goal) where $s_0 \in S$ is the **initial state**, *succ*: **S** → ℘(**S**) is the **successor function** that for some state s returns a set of states that we can **transition to** from s, and *goal*: **S** → {True, False} is a **predicate** (function that returns a boolean value) that for a given state s determines if s is a **goal state** or not (there can be multiple states that satisfy the goal predicate). A state space search (typically) ends as soon as any goal state is found.

State Space Search: Example

- **Q:** What sequence of moves leads from the initial state to the goal state?

$s_0 = \begin{array}{|c|c|c|}\hline 8 & & 7 \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}$

$succ\left(\begin{array}{|c|c|c|}\hline 8 & & 7 \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}\right) = \left\{ \begin{array}{|c|c|c|}\hline & 8 & 7 \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}, \begin{array}{|c|c|c|}\hline 8 & 7 & \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}, \begin{array}{|c|c|c|}\hline 8 & 5 & 7 \\ \hline 6 & & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array} \right\}$

$goal\left(\begin{array}{|c|c|c|}\hline 8 & & 7 \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}\right) = False$

$goal\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & \\ \hline\end{array}\right) = \textbf{True}$

$goal\left(\begin{array}{|c|c|c|}\hline 8 & 7 & \\ \hline 6 & 5 & 4 \\ \hline 3 & 2 & 1 \\ \hline\end{array}\right) = False$

initial state

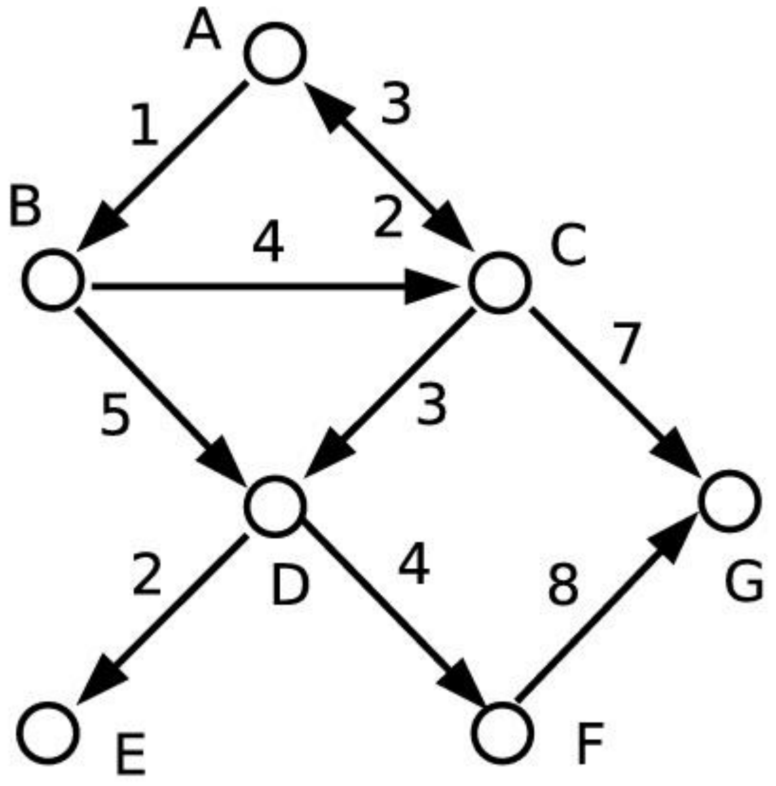
8		7
6	5	4
3	2	1

goal state

1	2	3
4	5	6
7	8	

State Space Search: **Graph**

- **State space search** amounts to a search through a **directed graph**
  - **Nodes** = states
  - **Edges** = transitions between states
- The graph is, however, not specified explicitly (nodes not „given in advance")
  - Graphs given **implicitly**
  - It may contain cycles (not a DAG!)
- If we also need/have transition costs, then it's a **weighted directed graph**
  - **Q:** how/why is this **different** than shortest paths problems (in directed graphs)?



State Space Search: **Tree**

- By **searching** through a directed graph, we gradually construct a search tree
  - **Q:** Do you know any graph search algorithms that create search trees? ☺
- We do this by **expanding** one node at a time
  - **Q:** this sounds familiar, no? ☺
  - **Caveat**: we typically don’t know the successors „out of the box”
    - Often need to **„compute”** the set of possible successor states **„on the fly”**, as we don’t know them in advance (large space of possible states)
    - Depending on the problem, *succ* function may not be trivial

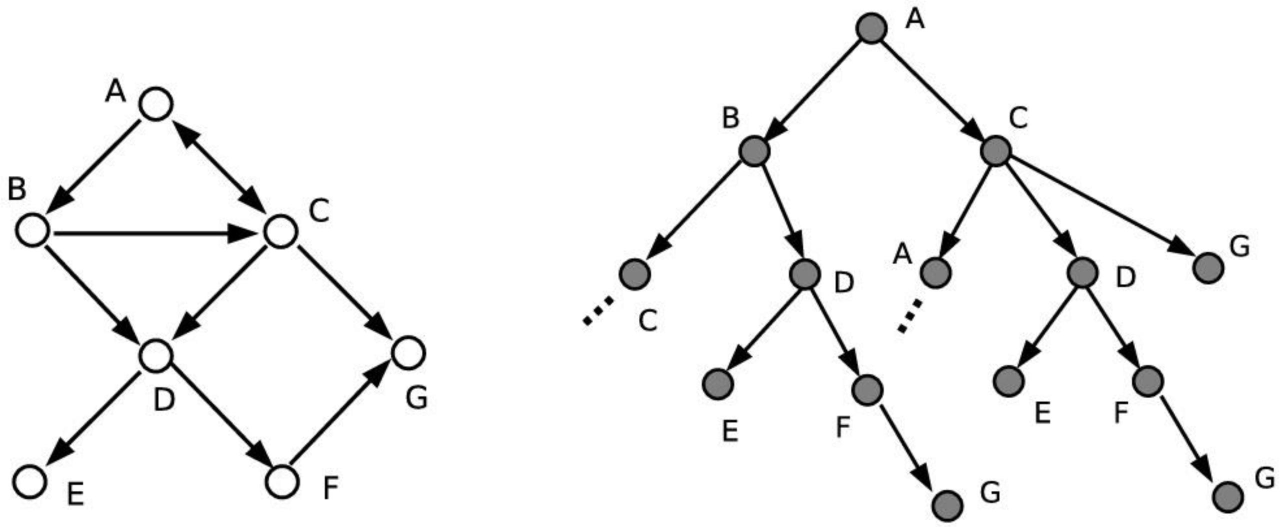
Search strategy: from graph to search tree

Search strategy

Search strategy defines the **order** in which the nodes are expanded. Different strategies yield different orders of states being visited.

- **Q:** Heard of any **search strategies** for graphs (when we start from a predefined „initial node”)? ☺
- **Q:** Does **revisiting** states

  makes sense?



State vs. Node

- For **efficient search**, we often need to store more than just the **state** in each **node** of the search tree
  - Especially true for informed SSS algorithms (which we cover next time)
- **Node n = (s, d)** is a data structure that stores the state s and the depth d of the node in the search tree
- We will define the corresponding functions that return s and d from n
  - *state*(n) = s
  - *depth*(n) = d
- And a function setting a state as **initial state** of the search
  - *init*(s) = (s, 0) (returns the **node** with **state** s and **depth** 0)

General Search Algorithm

- We define a general search algorithm
  - Think of it as **abstract search algorithm**
- Contains functions, whose concrete implementation depends on the choice of the actual search algorithm
- (Dynamic) Set of **open nodes** – nodes in the search tree that we reached: a „frontier” of the search tree
- **Generic (abstract) functions**:
  - *take*(*l*) – gets the next node from the set of open nodes *l*
  - *expand(n, succ)* – expands node n using *succ*
  - *insert(n, l) – adds node n to the list of open nodes l*

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

General Search Algorithm

- **Generic (abstract) functions**:
  - *take(l)* – gets the next node from the set of open nodes *l*
  - *expand(n, succ)* – expands node n using *succ*
  - *insert(n, l) – adds node n to the list of open nodes l*

- When **expanding** a node, we must update all components stored in it

```
search(s_0, succ, goal)
  open = [init(s_0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

```
expand(n, succ)
  sstates = succ(state(n))
  nodes = []
  for s in sstates
    nodes = nodes U (s, depth(n) + 1)
  return nodes
```

Content

- State Space Search: Fundamental AI Problem
- Uninformed Search
- Example: missionaries & cannibals

Search

- There are generally two types of search
- **Uninformed (blind) search**
  - No additional information about the problem, that could indicate whether one state is perhaps closer to the goal state than another state
- **Informed (directed, heuristic) search**
  - Additional information helps avoid some states and speed up the search
  - Problem-specific estimate of state's distance from the goal is available

Comparing SSS Problems and Algorithms

Properties of the problem

- **|S|** – number of states
- b – branching factor of the search tree (number of states that are reached from some state)
- d – the smallest depth at which we find a goal state
- m – maximum depth of the search tree (can be infinity)

Comparing SSS Problems and Algorithms

Properties of the search algorithms

- **Completeness** – an algorithm is **complete** if and only if (iff) it finds a solution (goal state) whenever a solution exists
- **Optimality** – an algorithm is optimal if and only if the solution it finds is optimal, i.e., if finds the goal state with the „smallest cost"
- **Time complexity** – runtime of the algorithm, corresponds to the number of generated nodes in the search tree
- **Space complexity** – memory space occupied by the algorithm, corresponds to the number of stored nodes (maximal length of *open*)

Example: 8-Puzzle Problem

- Number of states |S|?
  - How many different configs of the board?
  - Imagine the board is empty and you're „randomly" placing numbers one by one
- Minimal, maximal, and average b?
  - Minimal? If the „empty block" is in the corner
  - Maximal? If the „empty block" is in the middle
  - Average? What other positions of the „empty block" are there?

initial state

8		7
6	5	4
3	2	1

goal state

1	2	3
4	5	6
7	8	

Uninformed Search Strategies

- Breadth-First Search (BFS)
- Uniform Cost Search
- Depth-First Search (DFS)
- Depth-Limited Search
- Iterative Deepening Search

Breadth-First Search

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

- BFS is already our good friend ☺
- General search algorithm → BFS?
  - *open* needs to be a **queue**
  - *take(open)* is then **dequeue**(open)
  - *insert(m, open)* is then **enqueue**(m, open)
- Reminder: in BFS, any node at depth d+1 is expanded only after all nodes at depth d have been expanded

Breadth-First Search

- BFS is **complete**
  - If there is node n in the search tree of BFS such that ***goal***(***state***(n)) = True, BFS will clearly (eventually) **find** it

- BFS is **optimal**
  - **Reminder:** BFS reaches vertices of the graph in shortest possible paths from the source vertex!

```
bfs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = dequeue(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      enqueue(m, open)
  return False
```

Breadth-First Search

- **Time complexity (on the search tree)**
  - d as (minimal) depth of any goal node
  - 1 (root, s_0) + b (first level) + b^2 (second level) + … + b^d + b^{d+1} = $O(b^{d+1})$
  - Worst case: goal node **last** at the level d – this means that most nodes of the level d+1 (b^{d+1} - b) will be added to open

```
bfs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = dequeue(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      enqueue(m, open)
  return False
```

- **Space complexity: $O(b^{d+1})$**

Breadth-First Search

- **Space complexity: O(b^{d+1})**
  - Main shortcoming of BFS

- Example: b = 4, d = 16, and 10B/node
  - 4^17 * 10 B = ca. **43 GB** of memory

```
bfs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = dequeue(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      enqueue(m, open)
  return False
```

Uniform cost search

- **Uniform cost search** is a state-space-search algorithm for problems where there are **transitions costs**
  - For a **weighted state-transition graph**
  - **Q:** Why not just run shortest paths algorithms on graphs (e.g., Dijkstra)?

- **If transitions differ in costs, we need to have a few modifications**
  - succ needs to return also the cost of transition: succ: **S** $\rightarrow \wp($**S** $\times \mathbb{R}+)$
  - Nodes store the total cost c to reach them instead of depth d: n(s, c), c = cost(n)
  - *expand(n, succ)* needs to sum the cost (instead of increase depth)

```
expand(n, succ)
  sstates = succ(state(n))
  nodes = []
  for (s, c) in sstates
    nodes = nodes ∪ (s, cost(n) + c)
  return nodes
```

Uniform cost search

- **Uniform cost search** is a state-space-search algorithm for problems where there are **transitions costs**

- General search algorithm → UCS?
  - *open* needs to be a **priority queue**
  - *take(open)* is then **extract-min**(open)
  - *insert(m, open)* is then a **heap insertion**

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

Uniform cost search

- **Uniform cost search** is a state-space-search algorithm for problems where there are **transitions costs**

- General search algorithm →UCS?
  - *open* needs to be a **priority queue**
  - *take(open)* is then **extract-min**(open)
  - *insert(m, open)* is then a **heap insertion**

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

- **Reminder:** heap is the DS used for implementing a priority queue
  - Runtime of extract-min?
  - Runtime of insertion into heap?

Uniform Cost Search

- UCS is **complete**
  - If there is node n in the search tree of BFS such that ***goal***(***state***(n)) = True, BFS will clearly (eventually) **find** it
- UCS is **optimal**
  - **Q**: prove it!
  - Insertion of nodes into a **priority queue** (key = cost),
  - Therefore, when UCS reaches a goal node, it will be with minimal possible cost!

```
ucs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = extract-min(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open) # heap insertion
  return False
```

Uniform Cost Search

- **Time & space complexity**
  - Let C^* be the optimal (minimal) cost of reaching a goal node
  - Let ε be the minimal transition cost
  - The goal state is at depth $d = \lfloor C^*/\varepsilon \rfloor$
  - $\mathbf{O(b^{d+1}) = O(b^{\lfloor C^*/\varepsilon \rfloor + 1})?}$
  - But this **ignores** the cost of maintenance of the heap
    - Heap insert: **O(log n)** where n is the size of *open*
    - *„n"* = $\mathbf{b^{d+1} = b^{\lfloor C^*/\varepsilon \rfloor + 1}}$
  - **Runtime:** $\mathbf{O(b^{\lfloor C^*/\varepsilon \rfloor + 1} \log b^{\lfloor C^*/\varepsilon \rfloor + 1})}$

```
ucs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = extract-min(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)  # heap insertion
  return False
```

Depth-First Search

- DFS is also already our good friend ☺
- General search algorithm → DFS?
  - *open* needs to be a **stack**
  - *take(open)* is then **pop**(open)
  - *insert(m, open)* is then **push**(m, open)

```
search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      insert(m, open)
  return False
```

Depth-First Search

- DFS is not* **complete**
  - If there are cycles in the graph, DFS will result in an infinite loop
    - *Assuming we allow revisiting of the already visited states
- DFS is not **optimal**
  - **Reminder**: for DFS (unlike BFS) we don’t have a guarantee to have reached a state with minimal distance first time we discover it

```
dfs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = pop(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      push(m, open)
  return False
```

Depth-First Search

- m: the maximal depth of the search tree
- **Time complexity**
  - **O(b^m)**, which is pretty bad if the search tree is unbalanced and m >> d (depth where the goal node is)
  - We cannot really balance the search tree of a state space search graph ☹
    - It's defined by the problem
- **Space complexity**
  - **O(b*m)**
  - **Q:** Why?

```
dfs-search(s0, succ, goal)
  open = [init(s0)]
  while len(open) > 0
    n = pop(open)
    if goal(state(n))
      return n
    for m in expand(n, succ)
      push(m, open)
  return False
```

Depth-Limited Search

- Effectively, **limiting** our DFS to the **maximal depth**
- Let k be the maximal depth that DFS is allowed to reach
- Best we can do with DFS if we have a **runtime limitation** (which we always do :)

- DLS is **not complete**: it will not find a solution if the goal node is at depth $d > k$ (also not optimal, like DFS)
- **Time complexity** is $O(b^k)$
- **Space complexity** is $O(b*k)$

Depth-Limited Search

Useful if:

- **(1)** we know the depth d at which the goal state will appear **and**
- **(2)** that depth is not prohibitively large

- Then we, obviously, set k to d

```
limited-dfs-search(s0, succ, goal, k)
  open = [init(s0)]
  while len(open) > 0
    n = pop(open)
    if goal(state(n))
      return n
    if depth(n) < k
      for m in expand(n, succ)
        push(m, open)
  return False
```

Iterative Deepening Search

- **Iterative Deepening Search** is an iterative application of depth-limited search for increasing maximal allowed depth k
- **Tradeoff** between advantages of DFS and BFS
  - Completeness/Optimality of **BFS**
  - Space complexity of **DFS**
- Still, if the goal node is very deep in the search tree, no uninformed search algorithm will be efficient enough

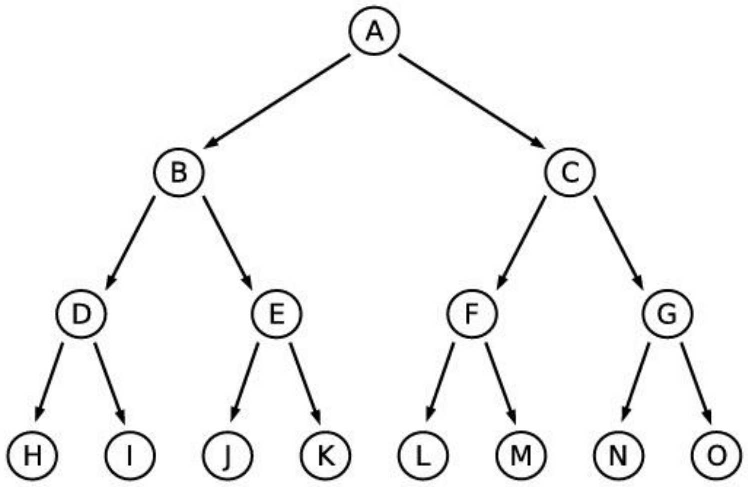
```
limited-dfs-search(s_0, succ, goal, k)
  open = [init(s_0)]
  while len(open) > 0
    n = pop(open)
    if goal(state(n))
      return n
    if depth(n) < k
      for m in expand(n, succ)
        push(m, open)
  return False
```

```
iter-deep-search(s_0, succ, goal)
   for k = 0 to inf
     n = limited-dfs-search
              (s_0, succ, goal, k)
     if n ≠ False
       return res
```

Iterative Deepening Search

- At first glance, IDS seems utterly **inefficient**: same nodes expanded many times over again
- This is typically **not a problem**: expansion is **more repeated** the shallower the node is in the search tree
- **Optimal** and **complete** (like BFS)
- **Time complexity**: **O(b^d)**
  - Like BFS, better than DFS
- **Space complexity**: **O(b*d)**
  - Like DFS, better than BFS

```
iter-deep-search(s0, succ, goal)
   for k = 0 to inf
     n = limited-dfs-search
             (s0, succ, goal, k)
     if n ≠ False
       return res
```



IDS: A | A, B, C | A, B, D, E, C, F, G | A, B, D, H, I, E, J, K, ...

Avoiding revisiting states

- If the search algorithm is **optimal** (BFS, IDS), there is no reason to allow repetition of states
- So we can keep track of **visited states** and avoid putting into *open* any node whose state has already been visited
- **Q:** what data structure to use for *visited*?
- If no state is ever repeated, complexity $\mathbf{O(b^{d+1})}$ reduces to $\mathbf{O(\min(b^{d+1}, b*|S|))}$
  - In most problems, $\mathbf{b*|S| < b^d}$

```
search(s0, succ, goal)
  open = [init(s0)]
  visited = []
  while len(open) > 0
    n = take(open)
    if goal(state(n))
      return n
    visited = visited U {state(n)}
    for m in expand(n, succ)
      if state(m) not in visited
        insert(m, open)
  return False
```

Content

- State Space Search: Fundamental AI Problem
- Uninformed Search
- Example: Missionaries & Cannibals

Missionaries and Cannibals

Missionaries & cannibals

N missionaries and N cannibals must be brought over by boat from one side of the river to the other. At no time should the missionaries be outnumbered by the cannibals on either side of the river. The boat can carry up to two passengers and cannot move by itself. We are looking for a solution with the **fewest possible number of steps**.

Missionaries and Cannibals

- Problem = (s_0, *succ*, *goal*)
- State representation? (m, c, position)
  - m – number of missionaries on the left river bank (N-m on the right then)
  - c – number of cannibals on the left river bank (N-c on the right then)
  - position of the boat – **L** (left river bank) or **R** (right river bank)

  - s_0 = (N, N, **L**)

  - State allowed (safe)?
    - *safe*(s) = **True** if ($m \geq c$ or $m = 0$) AND ($N-m \geq N-c$ or $m = N$)

  - *goal*: True **if** s = (0, 0, R) otherwise False

Missionaries and Cannibals

- Problem = (s_0, *succ*, *goal*)
- State representation? (m, c, position)
  - m – number of missionaries on the left river bank (N-m on the right then)
  - c – number of cannibals on the left river bank (N-c on the right then)
  - position of the boat – **L** (left river bank) or **R** (right river bank)

  - ***succ***(s = (m, c, pos))?
    - If **pos = L**: {(m-1, c, R), (m-2, c, R), (m-1, c-1, R), (m, c-1, R), (m, c-2, R)}
    - If **pos = R:** {(m+1, c, L), (m+2, c, L), (m+1, c+1, L), (m, c+1, L), (m, c+2,L)}
    - But only allowed states are kept, for which safe(s) = True

Recognizing a state space search problem

- Recognizing that a problem is a state-space-search problem is key – then we need to figure out the **suitable state representation** and *succ*

- *Have we described all that is relevant for the problem?*
- *Have we abstracted away the unimportant details?*
- *Do we generate all possible moves?*
- *Are all moves that we generate legal?*
- *Do we generate undesirable states?*
- *Would it perhaps be smarter to incorporate state validity check directly into the test predicate goal?*
- *What are the properties of our problem?* |S| =? b =? d =? m=?
- *Is this a difficult problem?*

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
أسئلة؟