THE UNIVERSITY OF QUEENSLAND
AUSTRALIA
CREATE CHANGE

COMP3702/COMP7702
Artificial Intelligence

Module 1: Search

Dr Archie Chapman

Semester 2, 2020

The University of Queensland
School of Information Technology and Electrical Engineering

Thanks to Dr Alina Bialkowski and Dr Hanna Kurniawati

Table of contents

Agent design components

Recall our goal: To build a useful, intelligent agent

To start with:

- Computers perceive the world using sensors.
- Agents maintain models/representations of the world and use them for reasoning.
- Computers can learn from data.

So, to achieve our goal, we need to define our "agent" in a way that we can program it:

- The problem of constructing an agent is usually called the **agent design problem**
- Simply, it's about defining the **components** of the agent, so that when the agent acts rationally, it will accomplish the task it is supposed to perform, and do it well.

Agent design components

The following **components** are required to solve an agent design problem:

- **Action Space** (A): The set of all possible actions the agent can perform.
- **Percept Space** (P): The set of all possible things the agent can perceive.
- **State Space** (S): The set of all possible configurations of the world the agent is operating in.
- **World Dynamics/Transition Function** ($T : S \times A \to S'$): A function that specifies how the configuration of the world changes when the agent performs actions in it.
- **Perception Function** ($Z : S \to P$): A function that maps a state to a perception.
- **Utility Function** ($U : S \to \mathbb{R}$): A function that maps a state (or a sequence of states) to a real number, indicating how desirable it is for the agent to occupy that state/sequence of states.

The agent design components

Recall:

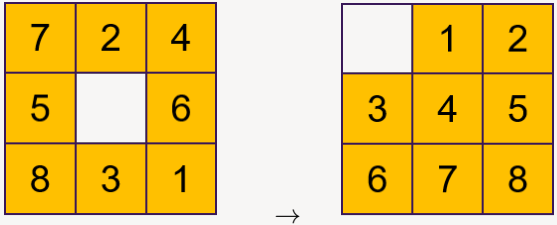
- Best action: the action that maximizes a given performance criteria
- A rational agent selects an action that it believes will maximize its performance criteria, given the available knowledge, time, and computational resources.

Utility function, $U : S \to \mathbb{R}$:

- A function that maps a state (or a sequence of states) to a real number, indicating how desirable it is for the agent to occupy that state/sequence of states.
- Crafting the utility function is a key step in the agent design process.

Example: 8-puzzle

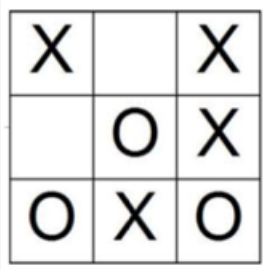
- Environment type: **Flat, discrete**
- Sensing uncertainty: **fully observable** vs partially observable
- Effect uncertainty: **deterministic** vs stochastic
- Number of agents: **single agent** vs multiple agents



Initial state Goal state

A classic search problem

Example: Noughts-and-crosses or Tic-tac-toe



- Environment type: **Flat, discrete**
- Sensing uncertainty: **fully observable** vs partially observable
- **Effect uncertainty**: deterministic vs **stochastic**
- Number of agents: single agent vs **multiple agents**

Adversarial search problem or zero-sum game - covered in Module 5

Introduction to search problems

Introduction to search

- In general the agent design problem is to find a mapping from sequences of percepts to action ($P \rightarrow A$) that maximises the utility function (U)
- Given the sequences of percepts or stimuli it has seen so far, what should the agent do next, so that the utility function can be maximized?
- Search is a way to solve this problem

When to apply search methods?

When we have:

- Sensing uncertainty: **fully observable**
- Effect uncertainty: **deterministic**
- Number of agents: **single agent**

then the agent design only needs to consider:

- **Action Space** (A)
- ~~Percept Space (P)~~
- **State Space** (S)
- **Transition Function** ($T : S \times A \rightarrow S'$)
- ~~Perception Function (Z)~~
- **Utility Function** ($U : S \rightarrow \mathbb{R}$) ...WHY?

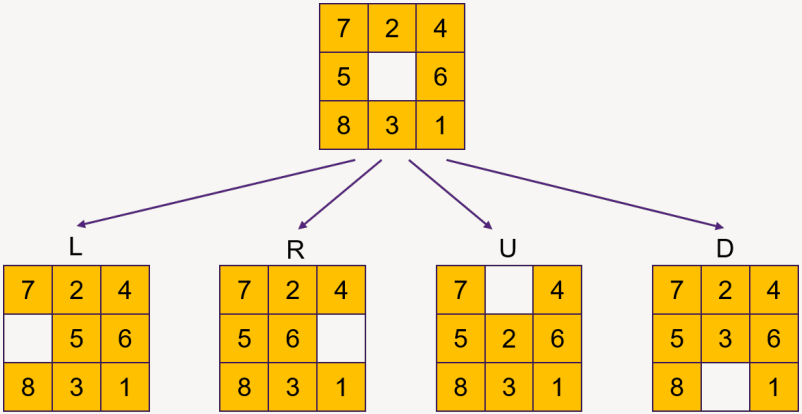
What is search?

A general framework to solve problems by exploring possibilities:

- Have multiple possible paths or options,
- Possibilities come from knowledge about the problem

What is search?

Using world dynamics, we can foresee future paths of different actions



Goal: How to find the solution with the smallest exploration cost?

Types of search methods

Two types:

- **Blind search**: Do not use any additional information to "guess" cost of moving from current node to goal
- **Informed search**: Use additional information to "guess" the cost of moving from current node to goal and decide where to explore next using this information

Formulating a problem as a search problem

Formulating a problem as a search problem

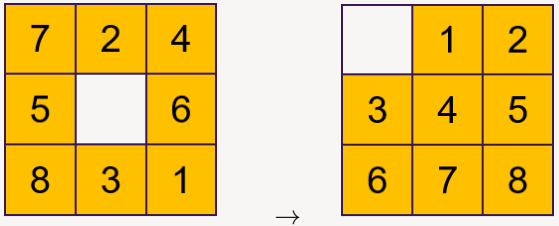
Problem: For a given problem/system, find a sequence of actions to move the agent from being in the initial state to being in the goal state, such that the utility is maximised (or cost of moving is minimised)

Design task is to map this problems to the agent components:

- **Action Space** (A)
- ~~Percept Space (P)~~
- **State Space** (S)
- **Transition Function** ($T : S \times A \to S'$)
- ~~Perception Function (Z)~~
- **Utility Function** ($U : S \to \mathbb{R}$)

Example: 8-puzzle

- **Action Space** (A): {up, down, left, right} for the empty cell
- **State Space** (S): All possible permutations
- **Transition Function** ($T : S \times A \rightarrow S'$): Given by tile moves, initial state is given
- **Utility**: $+1$ for goal state, 0 for all other states, i.e. agent's objective is to find a solution, goal state is given



Initial state → Goal state

How to do the search?

We want agent to be able to search for its own solution.

We need to embed this problem definition into a representation that is easy to search.

- Now: Discrete space
  - State graph representation
  - General structure of search algorithms
  - Uninformed (blind) search
  - Informed search
- Later: Continuous spaces

State graph representation

- A way to represent the problem concretely in programs
- Also a way of thinking about the problem
- We may or may not explicitly represent the state graph
- In problems with continuous or very large state space, state graph is often used as a compact representation of the state space (more on this later)

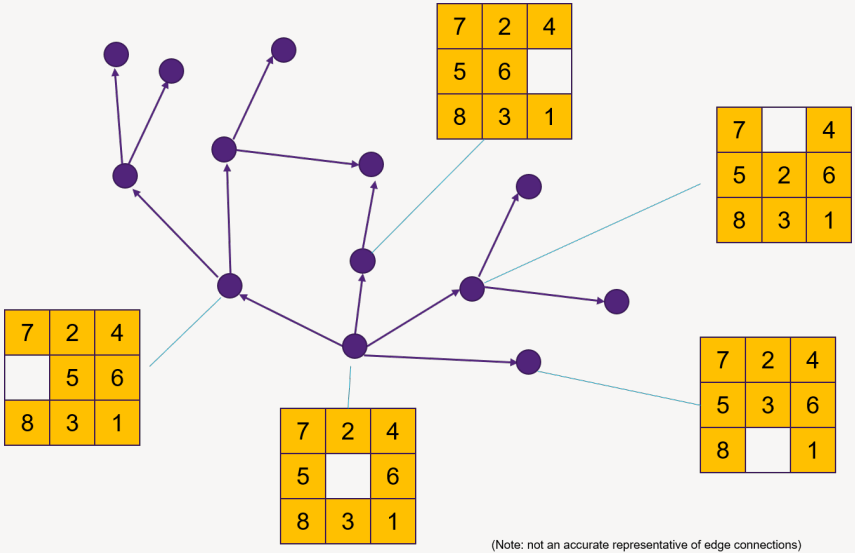
State graph representation

- **Graph**: (V, E)
- **Vertex** (V) represent states
- **Edges** (E) represent world dynamics

Each edge $\overline{ss'}$ is labelled with the cost to move from s to s'. It may also be labelled by the action to move from state s to s'.

- **Initial & goal state**: Initial & goal vertices
- A **solution** is a path from initial to goal vertices in the state graph
- **Cost**: the sum of the cost associated with each edge in the path
- **Optimal solution** is the shortest path

Example: 8-puzzle

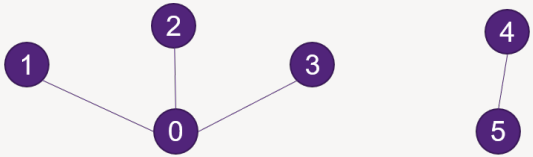


(Note: not an accurate representative of edge connections)

State graph representation

The state graph may have multiple connected components

- **Connected component:** sub-graph where there is at least one path in the sub-graph from each vertex in the sub-graph to any other vertex in the sub-graph.
- **Reachability:** If I'm in a particular state, can I reach another state?
- **Example**: In state graph representation of 8-puzzle, there are 2 connected components. If the initial & goal are in different connected component, there's no solution. We can check for this using a **parity check**, in Tutorial 2.



General structure of search algorithms

General structure of search algorithms

Put initial vertex in a "container" of states to be expanded

Loop:

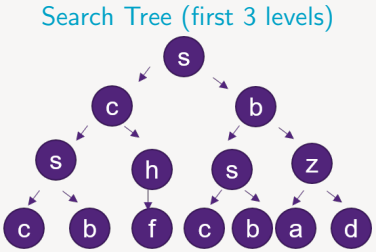
1. Select a vertex, v, from the "container"
2. If v is the goal vertex, then return
3. Expand v (i.e., put the results of `successor(v)` to the "container")

`successor(v)` is a function that:

- Takes a vertex v as input
- Outputs the set of immediate next vertices that can be visited from v (i.e., the end-point of out-edges from v)

Search tree

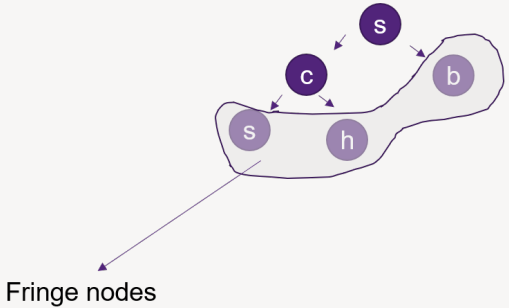
An abstract representation of the visited nodes (expanded + container):



If states can be revisited, the search tree may be infinite, even though the state graph (and state space) is finite.

The container is the fringe of the tree

Fringe or frontier: List of nodes in the search tree that have not been expanded yet



General structure of search algorithms with a search tree

Put the initial vertex as root of the search tree

Loop:

1. Select a node, t, from the fringe
2. If t is the goal vertex, then return
3. Expand t (i.e., put the results of `successor(t)` to the "container")

Consider `successor(t)` again: Now vertex v of the state graph corresponds to a node t in the search tree, so `successor(t)` are children of t in the search tree.

General structure of search algorithms with a search tree

Various search methods differ in:

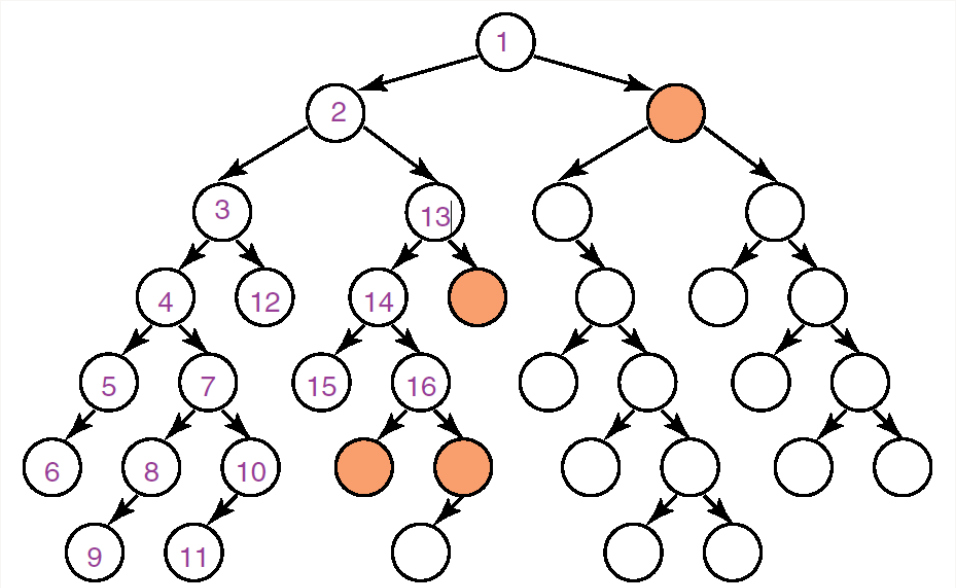
- Which node to expand next (i.e. retrieval order of the container), and
- How to expand the fringe.
- In Tutorial 2 (week 3) you will implement breadth-first search (BFS) and depth-first search (DFS).
- In the next lecture, we will examine their performance in terms of **completeness**, **optimality** and **complexity**.

Depth-first and Breadth-first Search

Depth-first Search

- **Depth-first search** treats the frontier as a stack
- It always selects one of the last elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$
  - p_1 is selected. Paths that extend p_1 are added to the front of the stack (in front of p_2).
  - p_2 is only selected when all paths from p_1 have been explored.

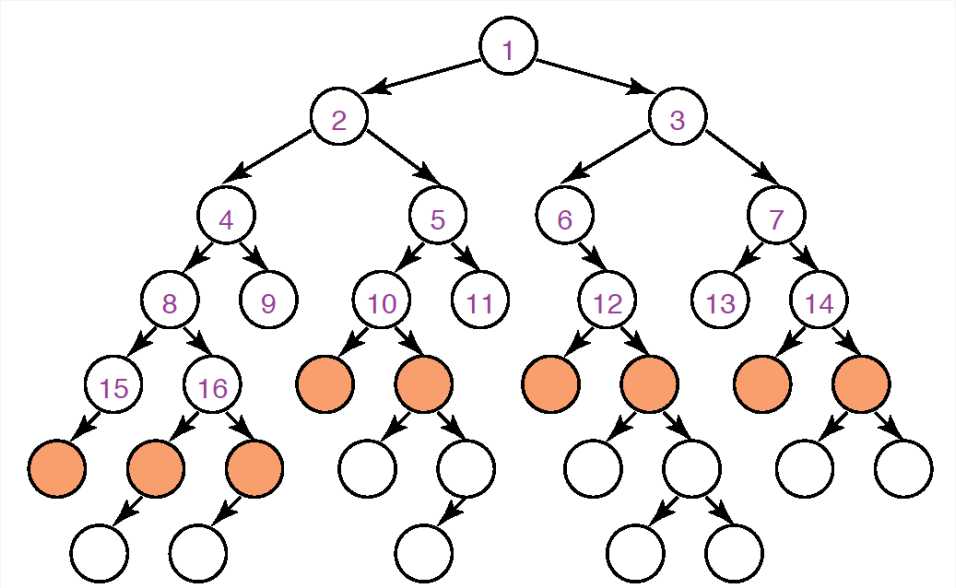
Illustrative Graph — Depth-first Search



Breadth-first Search

- **Breadth-first search** treats the frontier as a queue.
- It always selects one of the earliest elements added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_r]$:
  - p_1 is selected. Its neighbors are added to the end of the queue, after p_r.
  - p_2 is selected next.

Illustrative Graph — Breadth-first Search



More to come in Module 1...

Attributions and References

Thanks to Dr Alina Bialkowski and Dr Hanna Kurniawati for their materials.

Many of the slides in Module 0 are adapted from David Poole and Alan Mackworth, *Artificial Intelligence: foundations of computational agents*, 2E, CUP, 2017 `http://https://artint.info/`.

Other materials derived from Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3E, Prentice Hall, 2009.

All remaining errors are Archie's — please email if you find any: archie.chapman@uq.edu.au