COMP3702
Artificial Intelligence

Module 1: Search

Dr Alina Bialkowski

Semester 2, 2025

The University of Queensland
School of Electrical Engineering and Computer Science

Week 2: Logistics

- Assignment 0
  - Due: 1pm, Friday 15 August
  - Stats: 136 code & 63 report submissions / 574 students
- Assignment 1 will be released next week
- Ed Discussion board is active
- Scholarship Opportunities on BB

Week 2: Readings

- Russell & Norvig: Chapter 3.1-3.5
- Poole & Mackworth: Chapter 3.1-3.6

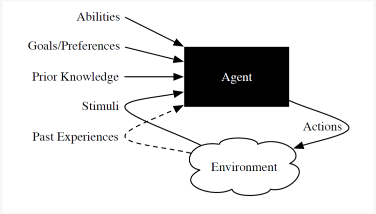
Table of contents

1. Agent design components
2. Introduction to search problems
3. Formulating a problem as a search problem
4. General structure of search algorithms
5. Blind search methods
6. Search with edge costs: Uniform cost search
7. Informed search
8. Greedy best-first search
9. A* Search

Agent design components

Intelligent agents

A computer **agent** gathers information about an environment, and takes actions autonomously based on that information.



- Perceives the environment (e.g. using sensors).
- Maintains models/representations of the environment and uses them for reasoning.
- Performs actions (e.g. via actuators) that change the state of the environment to achieve its goal.
- May learn from data.

Recall our goal: To build a useful, intelligent agent

In this class, goal is to design agents that act **rationally**.

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** ($T : S \times A \rightarrow S'$): A function that specifies how the configuration of the world changes when the agent performs actions in it.
- **Perception Function** ($Z : S \rightarrow P$): A function that maps a state to a perception.
- **Utility Function** ($U : S \rightarrow \mathbb{R}$): 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.

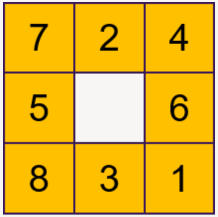
Defining the environment

Properties about the environment itself or the agent's knowledge about the environment:

- **Discrete** vs. **continuous**
  - Are the state / action / percept spaces finite?
- **Fully observable** vs. **partially observable**
  - Does the agent know the state of the world/itself exactly?
  - Is the percept function (Z) a bijection? [one-to-one mapping]
- **Deterministic** vs. **stochastic/non-deterministic**
  - Does the agent always know exactly which state it will be in after performing an action from a state?
  - Is the world dynamics (T) a function, i.e. does it map each (state, action) pair to exactly one next state?
- **Single agent** vs. **multiple agents**
  - Are there other agents interacting?
- **Static** vs. **dynamic**
  - Can the world change while the agent is "thinking"?

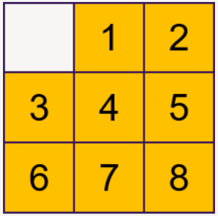
Example: 8-puzzle

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



Initial state

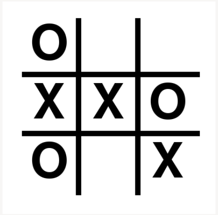
$\rightarrow$



Goal state

A classic search problem

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



- Environment type: **Flat, discrete, static**
- Sensing uncertainty: **fully observable** vs partially observable
- **Effect uncertainty**: deterministic vs **stochastic**
- Number of agents: single agent vs **multiple agents**
- Note: classification sometimes depends on time discretization
  - Here, 1 time step = a single move by the agent & the opponent

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:

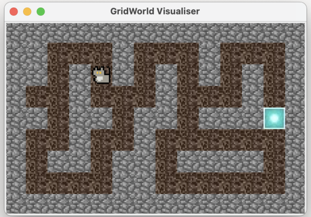
- **Action Space** (A)
- ~~Percept Space (P)~~
- **State Space** (S)
- **Transition Function** ($T : S \times A \rightarrow S'$)
- ~~Perception Function ($Z : S \rightarrow P$)~~
- **Utility Function** ($U : S \rightarrow \mathbb{R}$)

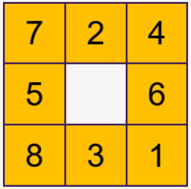
. . . WHY?

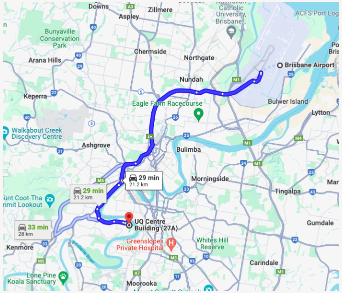
What is search?

A general framework to solve problems by exploring possibilities:

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

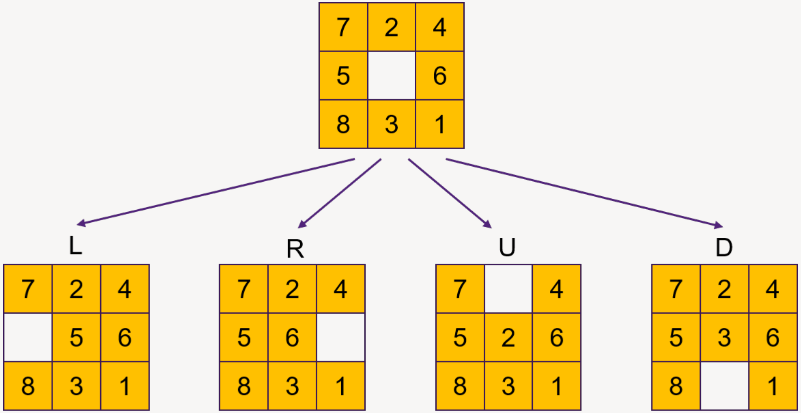






What is search?

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
    - DFS, BFS, iterative-deepening DFS, and uniform cost search (UCS)
- **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
    - Greedy best-first search and A* search

Formulating a problem as a search problem

Formulating a problem as a search problem

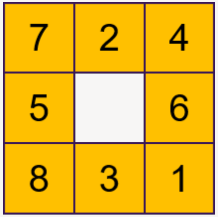
Problem: For a given problem/system, find a sequence of actions to move the agent from being in the initial state to being in the goal state, such that the utility is maximised (or cost of moving is minimised)

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

- **Action Space** (A)
- ~~Percept Space (P)~~
- **State Space** (S)
- **Transition Function** ($T : S \times A \rightarrow S'$)
- ~~Perception Function (Z)~~
- **Utility Function** ($U : S \rightarrow \mathbb{R}$) now becomes path cost function (aim to minimise)
- Initial & goal state

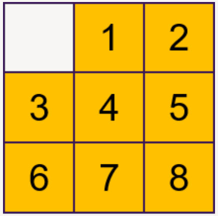
Example: 8-puzzle

- **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

$\rightarrow$



Goal state

How to do the search?

We want an 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.

We will mostly consider search in discrete space.

- Overview:
    - State graph representation
    - General structure of search algorithms
    - Uninformed (blind) search
    - Informed search

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)

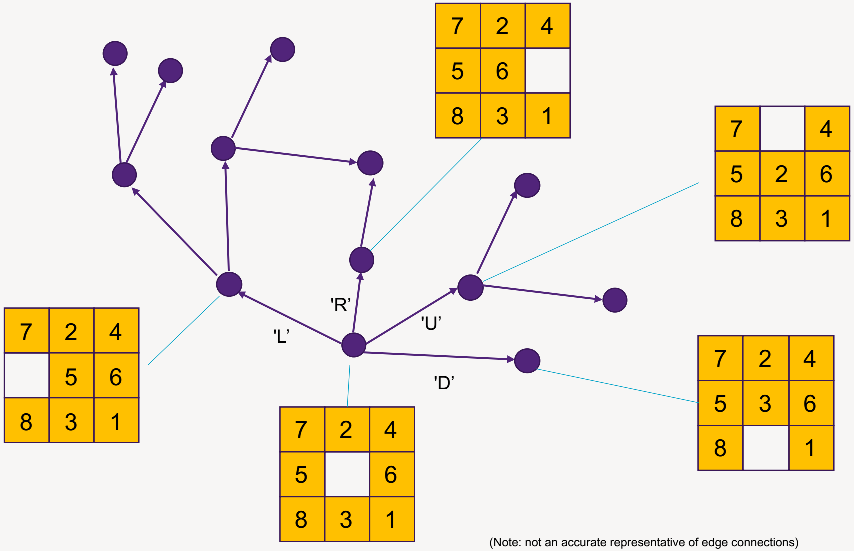
State graph representation

- **Graph**: (V, E)
- **Vertex** (V) (also called a node) represent states
- **Edges** (E) represent world dynamics (actions and their costs)

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 (lowest cost) path

Example: 8-puzzle

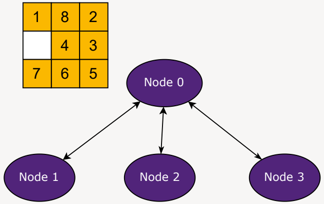


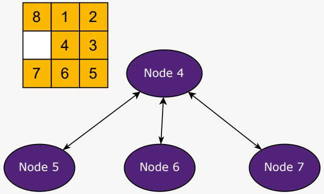
(Note: not an accurate representative of edge connections)

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 (node) 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.

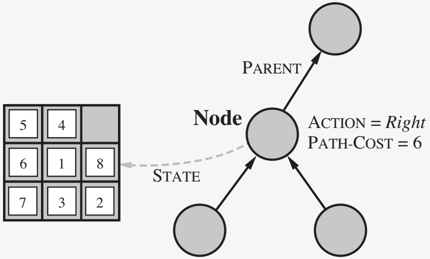




Node

A data structure that keeps track of

- a state
- a parent (the node that generated this node)
- an action (action applied to parent to get node)
- a path cost (from initial state to node)



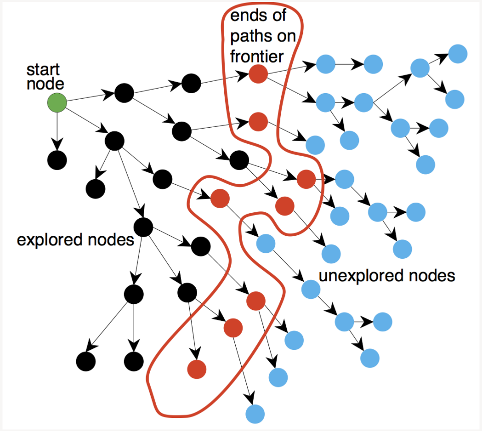
General structure of search algorithms

General structure of search algorithms

Generic search algorithm

- Given a graph, start and goal nodes, incrementally explore paths from the start nodes.
- Maintain a **frontier** (fringe) of paths from the start node that have been explored.
- As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered.
- The way in which the frontier is expanded defines the **search algorithm** or search strategy.

Generic search algorithm (P&M Figure 3.3)



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

General structure of search algorithms

Put initial node in the frontier

Loop:

1. If the frontier is empty, then no solution
2. Remove a node, *n*, from the frontier
3. If *n* contains the goal state, then return the solution
4. Expand *n* (i.e., adding the results of `successor(n)` into the frontier "container")

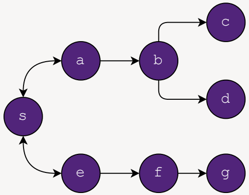
`successor(n)` is a function that:

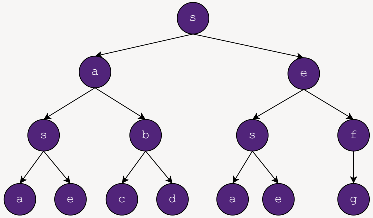
- Takes a node *n* as input
- Outputs the set of immediate next states that can be reached from *n*

Search tree

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

State Space Graph → Search Tree (first 3 levels)





- If states can be revisited, the search tree may be infinite, even though the state graph (and state space) is finite.
- We need to record the states we have already **visited** or **expanded** to avoid looping.

General structure of search algorithms with an explored set

Put initial node in the frontier
Create an empty explored set

Loop:

1. If the frontier is empty, then no solution
2. Remove a node, n, from the frontier
3. If n contains the goal state, then return the solution
4. Add the state to the explored set
5. Expand n (i.e., adding the results of `successor(n)` into the frontier "container"), if they aren't already in the frontier or explored set

`successor(n)` is a function that:

- Takes a node n as input
- Outputs the set of immediate next states that can be reached from n

General structure of search algorithms with a visited set

Put initial node in the frontier

Create a visited set, and add the initial state *[with 0 cost]*

Loop:

1. If the frontier is empty, then no solution
2. Remove a node, *n*, from the frontier
3. If *n* contains the goal state, then return the solution
4. Expand *n* (i.e., adding the results of `successor(n)` into the frontier "container" and the visited set), if they aren't already in the visited set *[or if the path cost to that state is less than when that state was previously visited]*

`successor(n)` is a function that:

- Takes a node *n* as input
- Outputs the set of immediate next states that can be reached from *n*

General structure of search algorithms

All search algorithms have the same basic structure, and differ only in which node is selected from the container (frontier) each loop. Sometimes, some differences in how states are tracked (explored vs visited) and when the goal check occurs.

- In Tutorial 2 (week 3) you will implement breadth-first search (BFS) and depth-first search (DFS).
- We will also examine their performance in terms of **completeness**, **optimality** and **complexity**.

Performance measures for search algorithms

Completeness

- **Complete**: The algorithm will find the solution whenever one exists.

Optimality

- **Optimal**: Return a minimum cost path whenever one exists.

Complexity

- **Time** (#steps) and **space** (memory) complexity
- Complexity analysis informs us of how the required time and memory needed to solve the problem increase as the input size increases
- **Input size**: Size of the state and action spaces of the search problem
- In state graph representation: Size of the graph
- Use computational complexity notation (e.g. Big-O)

Performance measures for search algorithms

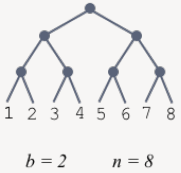
Big-O notation

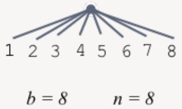
- Suppose $f(n)$ is the required time/space required to solve the problem if the input size is n
- Then, we say $f(n)$ is of complexity $O(g(n))$ if there is a constant k and n_0 such that:

$$0 \leq f(n) \leq k\, g(n) \quad \text{for all } n \geq n_0$$

  where n_0 is an arbitrary input size beyond which the highest order term $g(n)$ dominates the growth in required time or space, i.e. asymptotic analysis.
- Describes the upper bound of the algorithm's running time in the worst case.

Branching factor and depth: used to characterise graph topologies





Blind search methods

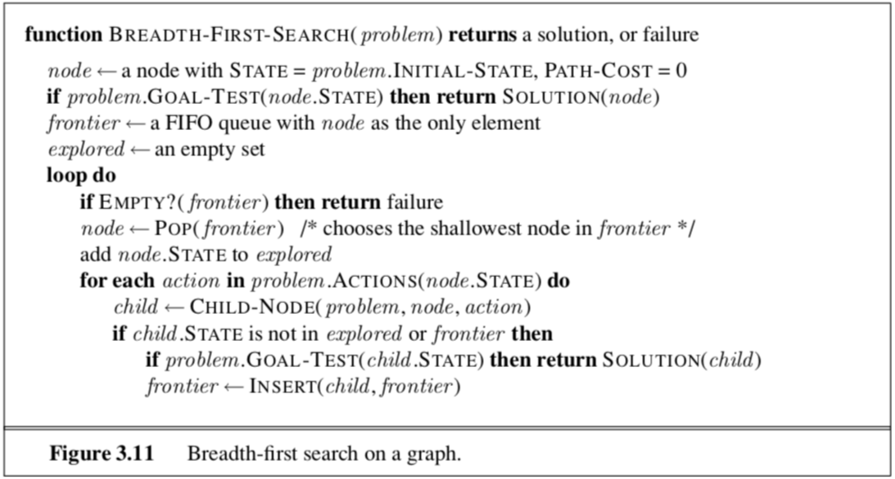
Blind search algorithms

- **Blind/uninformed search**: does not estimate the cost from the current node to the goal
- Examples:
  - Breadth first search (BFS)
  - Depth first search (DFS)
  - Iterative deepening DFS (IDDFS)
  - Uniform cost search (UCS)
- First 3 don't use any info about cost (unweighted graph)
- Uniform cost – uses edge weights

Breadth-first search

- **Breadth-first search** treats the frontier as a first-in first-out **queue**.
- It always selects one of the earliest elements added to the frontier (and hence the shallowest nodes).
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_n]$:
  - p_1 is selected. Its neighbours are added to the end of the queue, after p_n.
  - p_2 is selected next.

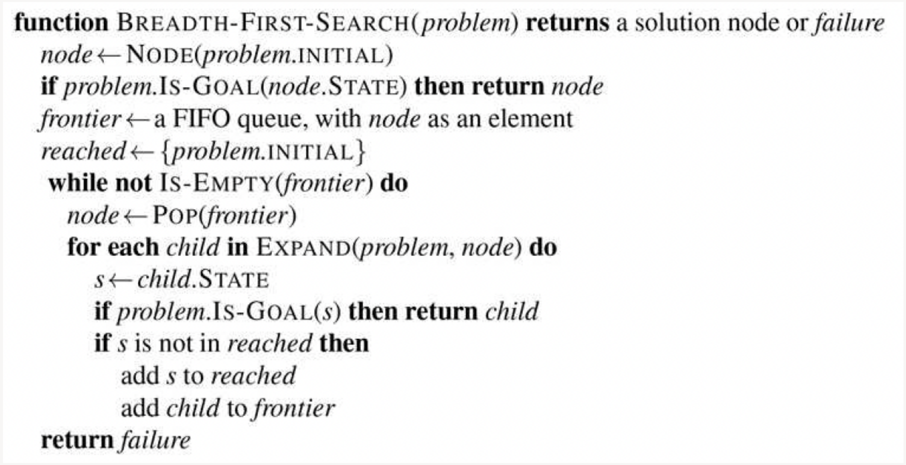
Breadth-first search algorithm (from R&N 3Ed)



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Breadth-first search algorithm (from R&N 4Ed)

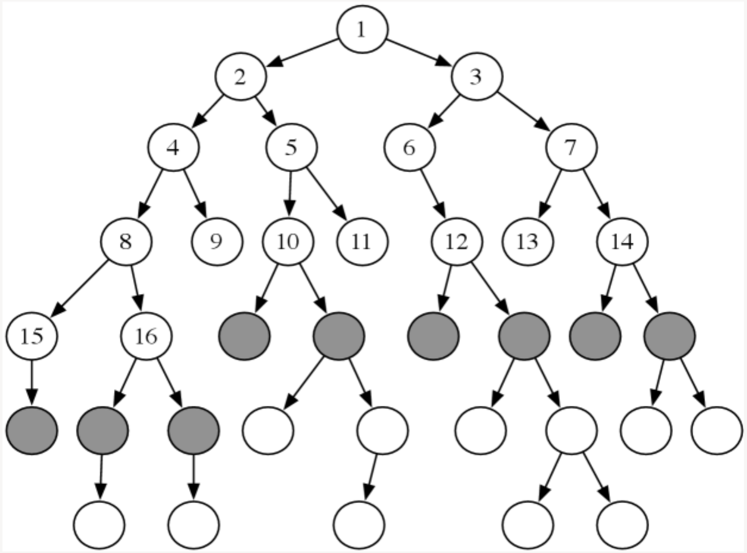


```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

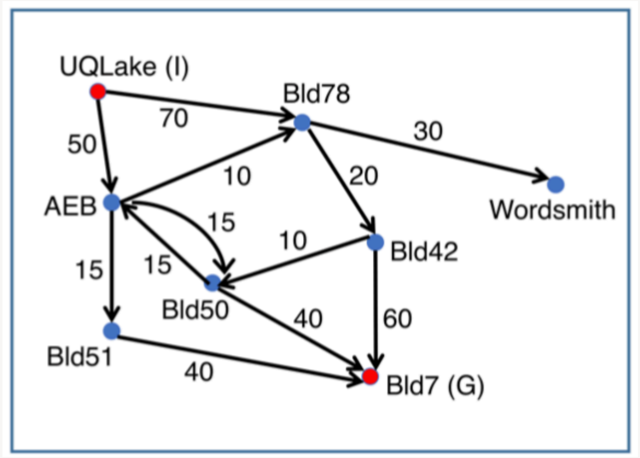
Breadth-first search python code (from R&N 4Ed)

```
def breadth_first_search(problem):
    #Search shallowest nodes in the search tree first.
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
    visited = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in visited:
                visited(s)
                frontier.appendleft(child)
    return failure
```

Illustrative Graph — Breadth-first Search



Example — Navigating UQ



Ignore costs on these edges

Breadth-first search: Properties and analysis

Parameters: b, branching factor; d, depth of shallowest goal node

Complete? Will BFS find a solution?

- Complete if b is finite

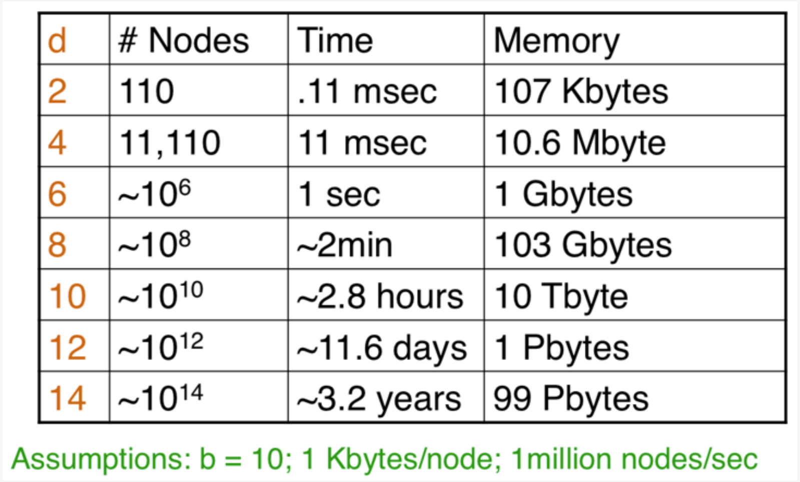
Generate optimal solution? Does BFS guarantee finding the path with fewest edges?

- **Yes** in #steps

Complexity:

- **Time**: $O(b^d) \quad \Leftarrow \quad 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1-1}}{b-1}$
- **Space**: $O(b^d)$ nodes to remember: $O(b^{d-1})$ explored nodes $+$ $O(b^d)$ unexplored nodes
- **Finds minimum step path, but requires exponential space!**

Let's get some intuition: Practical numbers for BFS

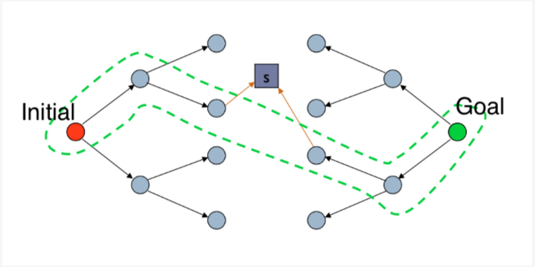


d	# Nodes	Time	Memory
2	110	.11 msec	107 Kbytes
4	11,110	11 msec	10.6 Mbyte
6	$\sim 10^6$	1 sec	1 Gbytes
8	$\sim 10^8$	~2min	103 Gbytes
10	$\sim 10^{10}$	~2.8 hours	10 Tbyte
12	$\sim 10^{12}$	~11.6 days	1 Pbytes
14	$\sim 10^{14}$	~3.2 years	99 Pbytes

Assumptions: b = 10; 1 Kbytes/node; 1million nodes/sec

Bidirectional Strategy

- 2 search trees and hence 2 fringe queues:

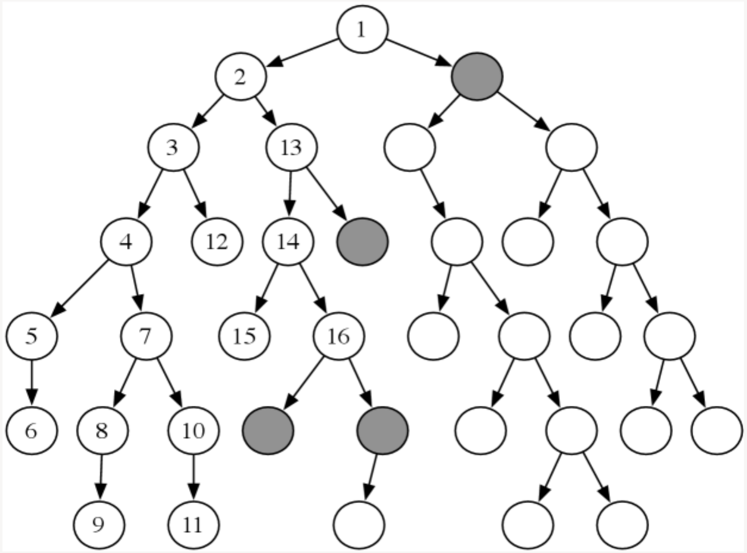


- Time and space complexity is : $O(b^{d/2}) << O(b^d)$

Depth-first search

- **Depth-first search** treats the frontier as a **stack**, or a last-in first-out queue.
- It always selects one of the last elements added to the frontier. i.e. the most recently added.
- If the list of paths on the frontier is $[p_1, p_2, \ldots, p_{n-1}, p_n]$
    - p_n is selected. Paths that extend p_n are added to the end of the stack.
    - p_{n-1} is only selected when all paths from p_n have been explored.

Illustrative Graph — Depth-first Search



Depth-first search: Properties and analysis

Parameters: b, branching factor; m, maximum depth; d, depth of shallowest goal node

Complete? Will DFS find a solution?

- Complete, if m and b are finite and nodes are not revisited

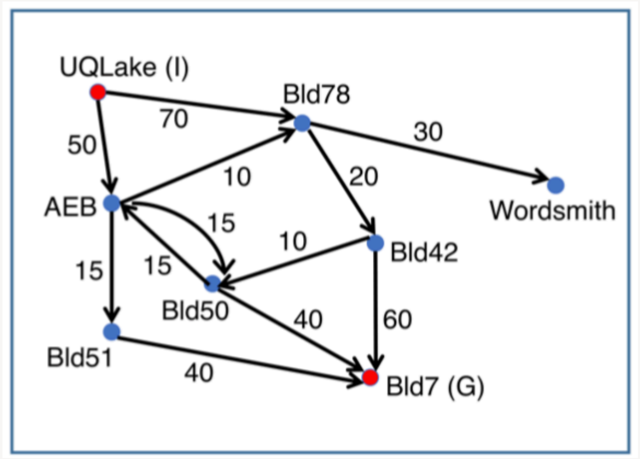
Generate optimal solution? Does DFS guarantee finding the path with fewest edges?

- No

Complexity:

- **Time**: $O(b^m) \quad \Leftarrow \quad 1 + b + b^2 + \ldots + b^m = \frac{b^{m+1-1}}{b-1}$
- **Space**: Can be implemented using $O(bm)$, or $O(m)$ using *backtracking* DFS. But be careful of revisiting states!
- **Efficient in use of space**

Example — Navigating UQ



Ignore costs on these edges

Iterative deepening DFS (IDDFS)

BFS: Finds minimum step path, but requires exponential space.

DFS: Efficient in space, but no path length guarantee.

Iterative deepening DFS

- Multiple DFS with increasing depth-cutoff until the goal is found.
- For $k = 1, 2, \ldots$: Perform DFS with depth cutoff k.
- Only generates nodes with depth $\leq k$.

IDDFS: Properties and analysis

Parameters: b, branching factor; m, maximum depth; d, depth of shallowest goal node

Complete? Will IDDFS find a solution?

- Complete if b is finite

Generate optimal solution? Does IDDFS guarantee to find the path with fewest edges?

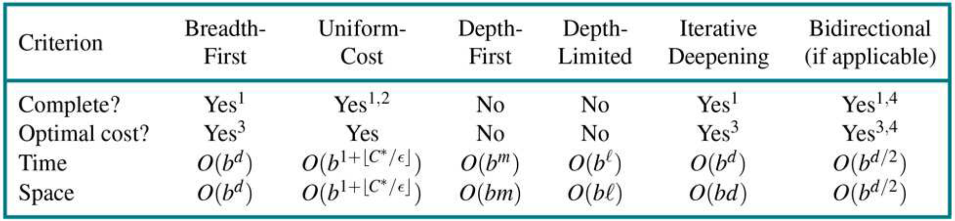
- **Yes** in #steps

Complexity:

- **Time**: $O(b^d) \quad \Leftarrow \quad db + (d-1)b^2 + \ldots + (1)b^d$
- **Space**: $O(bd)$
- **Finds minimum step path, and doesn't require exponential space!**

Comparing uninformed search algorithms

- Below table compares uninformed search algorithms
- b is the branching factor, m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit



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

Source: R&N 4Ed Fig 3.15

Search with edge costs: Uniform cost search

Search with edge costs: Uniform cost search

- Sometimes there are costs associated with edges.
- The **cost** of a path is the sum of the costs of its edges:

$$cost(n_0, \ldots, n_k) = \sum_{i=1}^{k} cost(n_{i-1}, n_i)$$

- An optimal solution is one with minimum cost.

Uniform cost search

- At each stage, uniform-cost search selects a path on the frontier with lowest cost.
- The first path to a goal is a least-cost path to a goal node.
- When edge costs are equal $\Rightarrow$ breadth-first search.

Uniform cost Search

- **Uniform cost search** treats the frontier as a **priority queue** ordered by path cost.
- It always selects one of the highest-priority vertices added to the frontier.
- If the list of paths on the frontier is $[p_1, p_2, \ldots]$:
    - p_1 is selected to be expanded.
    - Its successors are inserted into the priority queue.
    - The highest-priority vertex is selected next (and it might be a newly expanded vertex).

UCS: Properties and analysis

Parameters: b, branching factor; m, maximum depth; d, depth of shallowest goal node

$C*$: Cost of optimal solution, ϵ: minimum cost of a step

Complete? Will UCS find a solution?

- Complete if b is finite and all edges have a cost $\geq \epsilon > 0$

Generate optimal solution? Does UCS guarantee to find the path with **the lowest cost**?

- **Yes** if all edges have positive cost

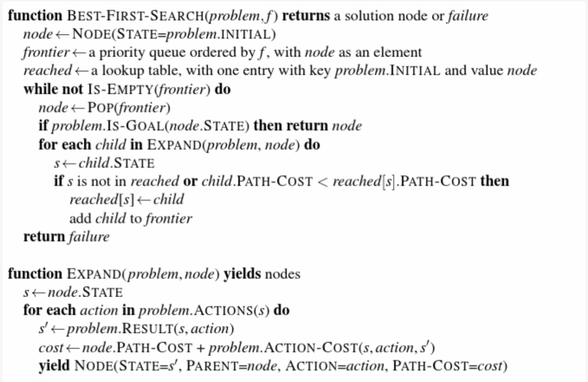
Complexity:

- **Time** and **Space**: $O(b^{1+\lfloor \frac{C*}{\epsilon} \rfloor})$

Best-first-search pseudocode, priority queue ordered by $f(n)$ (from R&N 4Ed)

General algorithm for search algorithms with a priority queue.

The priority, $f(n)$ can be a combination of path-cost from root node to current node: $g(n)$ and/or a heuristic which estimates the cost from the current node to the goal: $h(n)$



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

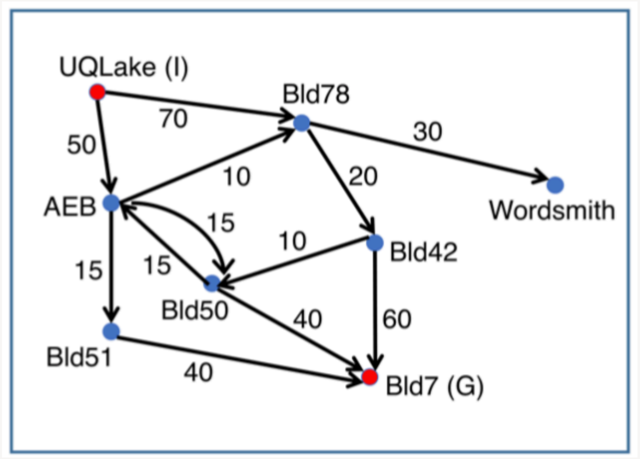
Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

Source: R&N Figure 3.7

Best-first-search python code, priority queue ordered by $f(n)$ (from R&N 4Ed)

```python
def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure
```

Example — Navigating UQ



Blind search: Summary

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

Informed search

Blind vs informed search

- **Blind search algorithms** (e.g. UCS) use only $g(n)$, the cost from the root to the current node to prioritise which nodes to search
- **Informed search algorithms** rely on **heuristics**, $h(n)$ that give an estimated cost from the current node to the goal to prioritise search
- In general, informed search is faster than blind search
- However, it is more difficult to prove properties of informed search algorithms, as their performance highly depends on the heuristics used

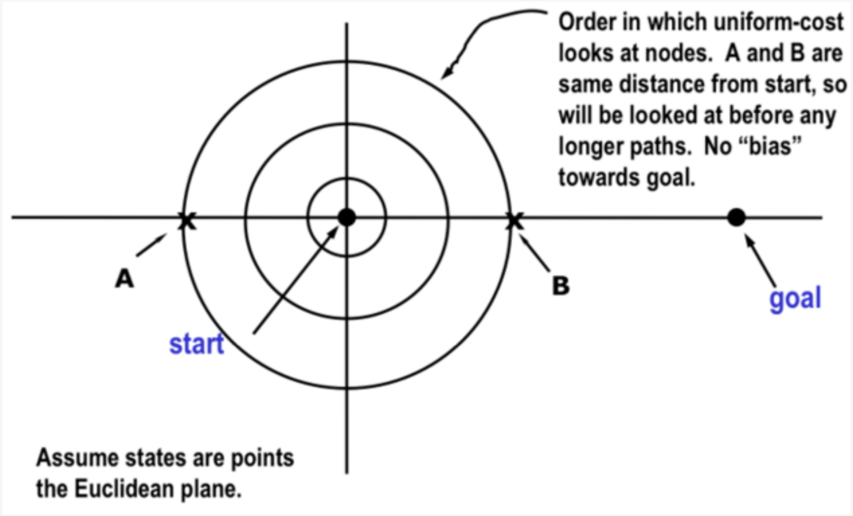
Blind vs informed search

- Informed search: Select which node to expand based on a function containing the estimated cost from the current node to the goal state
- Cost: $f(n) = g(n) + h(n)$
    - $g(n)$: Cost from root to node n
    - $h(n)$: Estimated cost from n to goal (usually based on heuristics)
    - In informed search, the node is selected based on f(n), and f(n) must contain h(n)

Informed search algorithms

- Greedy best-first search
- A* search

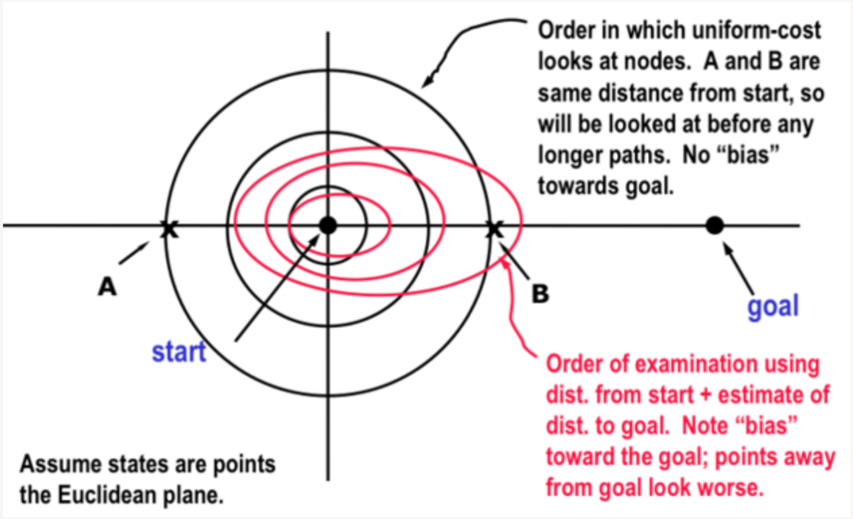
Blind vs informed search



Order in which uniform-cost looks at nodes. A and B are same distance from start, so will be looked at before any longer paths. No "bias" towards goal.

Assume states are points the Euclidean plane.

Blind vs informed search


Order in which uniform-cost looks at nodes. A and B are same distance from start, so will be looked at before any longer paths. No "bias" towards goal.
A
B
goal
start
Order of examination using dist. from start + estimate of dist. to goal. Note "bias" toward the goal; points away from goal look worse.
Assume states are points the Euclidean plane.

Informed search using heuristics

Idea: Don't ignore the goal when selecting paths.

- Often there is extra knowledge that can be used to guide the search: **heuristics.**
- $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- $h(n)$ needs to be efficient to compute.
- h can be extended to paths: $h(n_0, \ldots, n_k) = h(n_k)$.
- $h(n)$ does not overestimate, if there is no path from n to a goal with cost less than $h(n)$.
- An **admissible heuristic** is a heuristic function that *never overestimates* the actual cost of a path to a goal (it is *optimistic*).

Example heuristic functions

- If the nodes are points on a Euclidean plane and the cost is the distance, $h(n)$ can be the straight-line distance from n to the closest goal (i.e. ignoring obstacles).
- If the nodes are locations and cost is *time*, we can use the distance to a goal divided by the maximum speed (underestimate).
- If the goal is complicated, simple decision rules that return an approximate solution and that are easy to compute can make for good heuristics
- A heuristic function can be found by solving a simpler (less constrained) version of the problem.

Greedy best-first search

Greedy best-first search

Greedy Best-First Search has some similarities to UCS, but with some key differences:

- Also uses a **priority queue** to order expansion of fringe nodes, but utilises $h(n)$ [instead of $g(n)$ in UCS] as the priority

- The highest priority in priority queue for greedy best-first search is the node with the smallest **estimated cost** from the current node to the goal

- The estimated cost-to-goal is given by the heuristic function, $h(n)$

- If the list of paths (p_i) on the frontier, together with their cost (c_i) is $\{p_1 : c_1, p_2 : c_2, \ldots\}$:

    - p_1 is selected to be expanded.
    - Its successors are inserted into the priority queue.
    - The highest-priority vertex is selected next (and it might be a newly expanded vertex).

Greedy best-first search: Properties and analysis

Complete? Will greedy best-first search find a solution?

- **No** (it depends on the heurisitc)

Generate optimal solution? Is greedy best-first search guaranteed to find the lowest cost path to the goal?

- **No**

Complexity:

- Depends highly on the heuristic
- Worst case if the tree depth is finite: $O(b^m)$ where b is branching factor and m is maximum depth of the tree (i.e. it can be exponentially bad).

Example — Navigating UQ

Heuristic values (to Bld7)

h(UQLake) = 100

h(Bld78) = 50

h(AEB) = 53

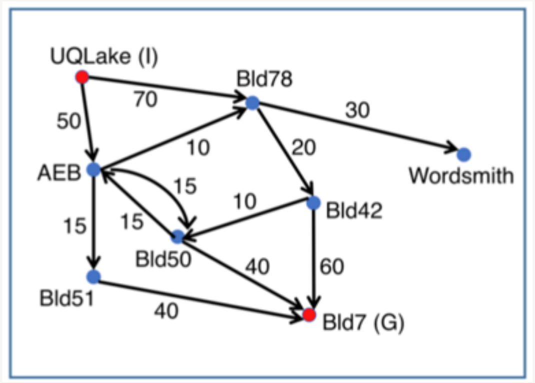
h(Wordsmith) = 1000

h(Bld42) = 50

h(Bld50) = 38

h(Bld51) = 30

h(Bld7) = 0



A* Search

A* Search

A* search uses both path cost and heuristic values

- It is a mix of uniform-cost and greedy best-first search
- $g(p)$ is the cost of path p from initial state to a node (UCS)
- $h(p)$ estimates the cost from the end of p to a goal (GBFS)
- A* uses: $f(p) = g(p) + h(p)$

$f(p)$ estimates the **total** path cost of going from a start node to a goal via p

$$\underbrace{\underbrace{start \xrightarrow{\text{path } p} n}_{g(p)} \underbrace{\xrightarrow{\text{estimate}} goal}_{h(p)}}_{f(p)}$$

A* Search Algorithm

- A* is a mix of uniform-cost and greedy best-first search, algorithmically similar to both
- It treats the frontier as a **priority queue** ordered by $f(p)$
- Highest priority is the node with the lowest f value The function $f(p)$ is the shortest path length from root to the node ($g(p)$) plus the estimated future reward from node p to the goal ($h(p)$)
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node

Example — Navigating UQ

Heuristic values (to Bld7)

h(UQLake) = 100

h(Bld78) = 50

h(AEB) = 53

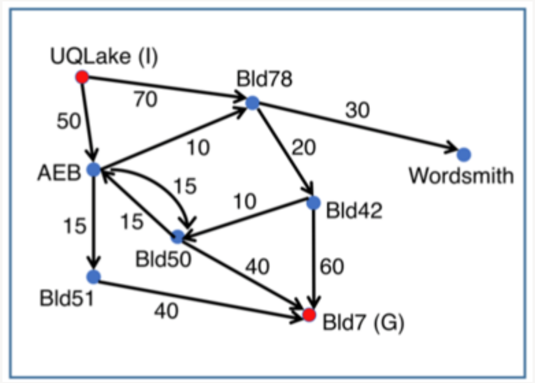
h(Wordsmith) = 1000

h(Bld42) = 50

h(Bld50) = 38

h(Bld51) = 30

h(Bld7) = 0



A* Search: Properties and analysis

- Will A* search find a solution?
- Is A* search guaranteed to find the shortest path?
- What is the time complexity as a function of length of the path selected?
- What is the space complexity as a function of length of the path selected?
- How does the goal affect the search?

Admissibility of A*

If there is a solution, A* always finds an optimal solution — as the first path to a goal selected — if the following conditions are met:

1. the search graph branching factor b is finite
2. edge costs are bounded above zero (there is some $\epsilon > 0$ such that all of the edge costs are greater than ϵ), and
3. $h(n)$ is $>= 0$ and does not overestimate the cost of the shortest path from n to a goal node.

. . . *we have seen 2 and 3 before:*

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

Why is A* admissible?

If a path p to a goal is selected from a frontier, can there be a shorter path to a goal?

- Suppose path p' is on the frontier.
- Because p was chosen before p', and $h(p) = 0$:

$$g(p) \leq g(p') + h(p').$$

- Because h is an underestimate:

$$g(p') + h(p') \leq g(p'')$$

  for any path p'' to a goal that extends p'.

So $g(p) \leq g(p'')$ for any other path p'' to a goal.

Why is A* admissible? extra working

Since p'' extends on p', and p'' is a path to the goal, we know that:

- $g(p') + \text{true cost to the goal} = g(p'')$
- Let's call the true cost from p' to the goal 'c'
- h is an estimate of the remaining cost to the goal, i.e. it is an estimate of c. Because h does not overestimate, that means that $h \leq c$.
- i.e. $g(p') + h(p') \leq g(p'')$
- Then joining the 2 equations we get the last line, that $g(p) \leq g(p'')$ for any other path p'' to a goal

How do good heuristics help?

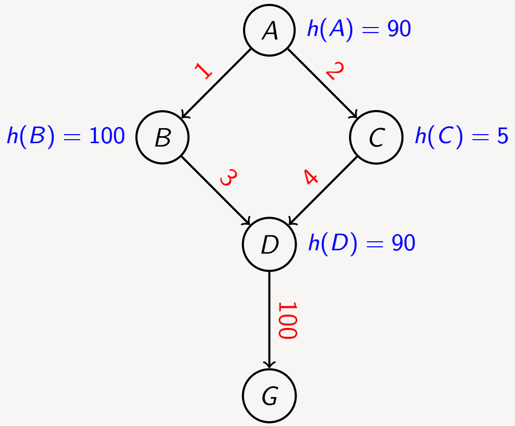
Suppose c is the cost of an optimal solution. What happens to a path p where

- $g(p) + h(p) < c$
- $g(p) + h(p) = c$
- $g(p) + h(p) > c$

How can a better heuristic function help?

Revisiting nodes

Optimality condition



Revisiting nodes

- Naive: Revisit all vs Discard all
  - Optimality vs complexity
- Discard a revisited node if the cost to the node via this new path is more than the cost of reaching the node via a previously found path.
  - The solution will be optimal, but may still be quite inefficient, due to revisiting nodes that are not in the optimal path.
- Works with a **consistent** (monotonic) heuristic
  - $h(n) \leq c(n, a, n') + h(n')$

Consistent heuristic

- $h(n)$ is **consistent** if for every node (n) and every successor (n') of n by any action (a), the estimated cost to reach the goal from n is not greater than the step cost of getting to n' + estimated cost of n' to the goal
- Also called the **triangle inequality**: If the heuristic h is **consistent**, then the number $h(n)$ will be less than the sum of the cost $c(n, a, n')$ of the action from n to n' plus the heuristic estimate $h(n')$:
  - i.e. $h(n) \leq c(n, a, n') + h(n')$

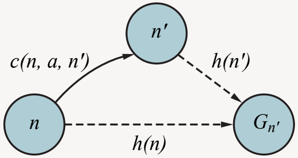


Figure 1: Consistent heuristic (Source: R&N 4e, p106)

Consistent heuristic

- If the heuristic is consistent, when A* expands a node n, the path to n is optimal
- Therefore, we don't need to revisit nodes that have been expanded
- Consistent $\rightarrow$ admissible, but not the other way around

How to generate heuristics

- Information about the problem (domain knowledge)
- Solve relaxed problem
- Knowledge about the sub-problems (e.g. solve tiles 1,2,3,4 in 8-puzzle)
- Learn from prior results of solving the same or similar problems
- Pre-computation (caching)

- Examples?
    - Euclidean distance
    - Hamming distance
    - Manhattan distance
    - Number of inversions in the 8-puzzle

Attributions and References

Thanks to A/Prof. Archie Chapman and Prof. Hanna Kurniawati for their materials.

Many of the slides are adapted from David Poole and Alan Mackworth, *Artificial Intelligence: foundations of computational agents*, 3E, CUP, 2023 `https://artint.info/`. These materials are copyright © Poole and Mackworth, 2023, licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Other materials derived from Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4E, Prentice Hall, 2022.