CS5011 Introduction to Search

Dr Alice Toniolo

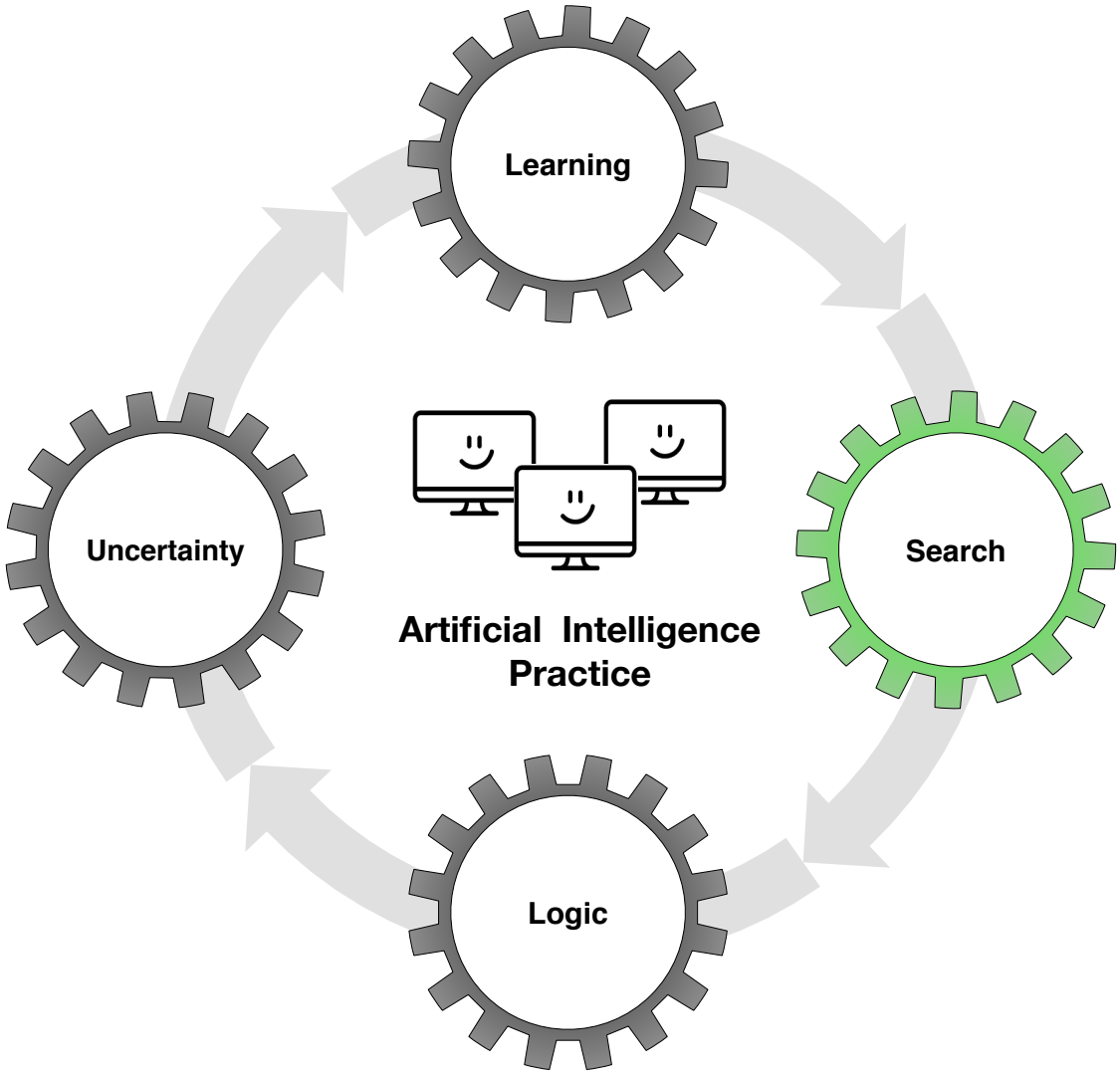
a.toniolo@st-andrews.ac.uk
JC 1.11

University of St Andrews

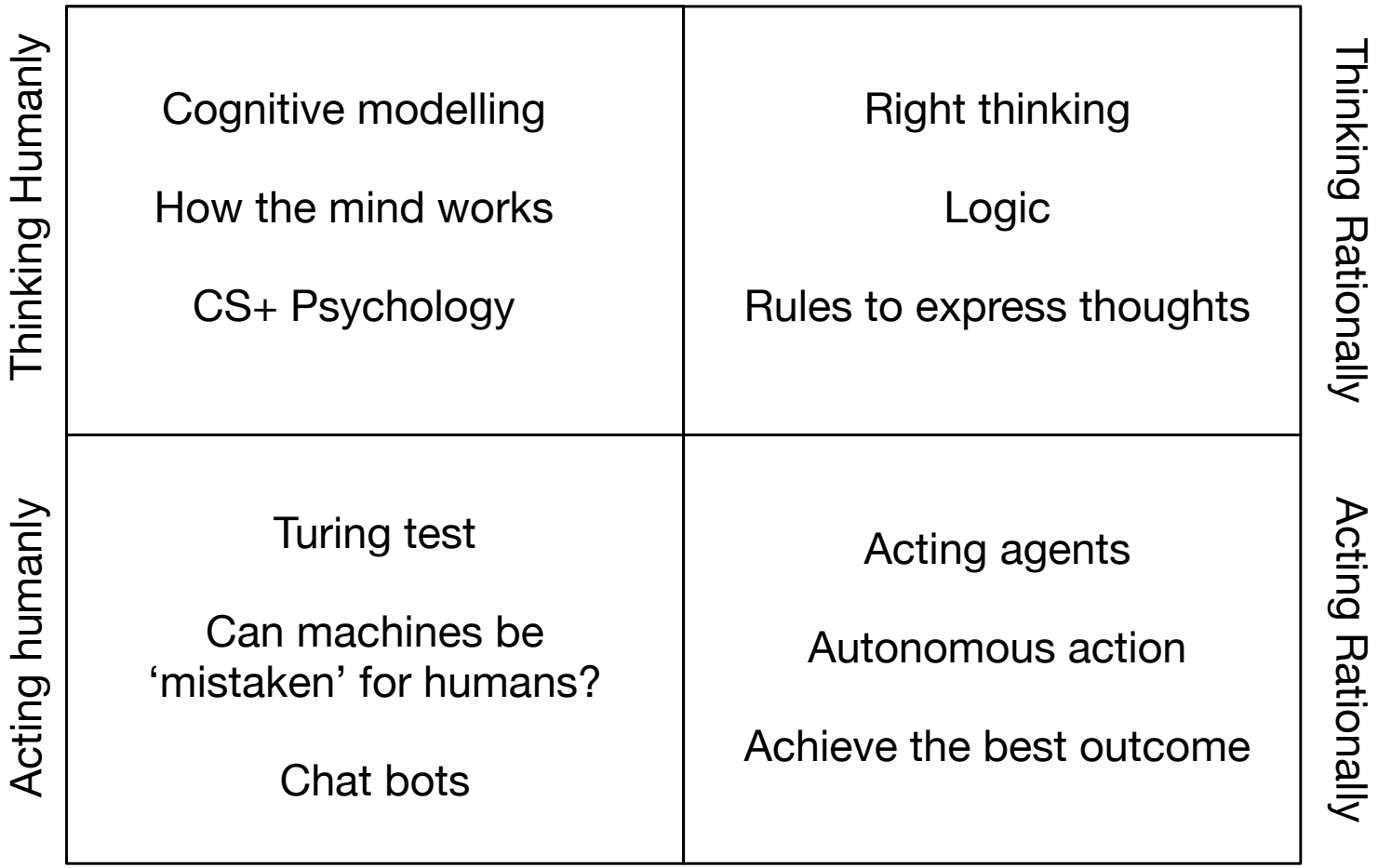
09-10-2017

Where are we?

How to search for a solution to solve a problem?



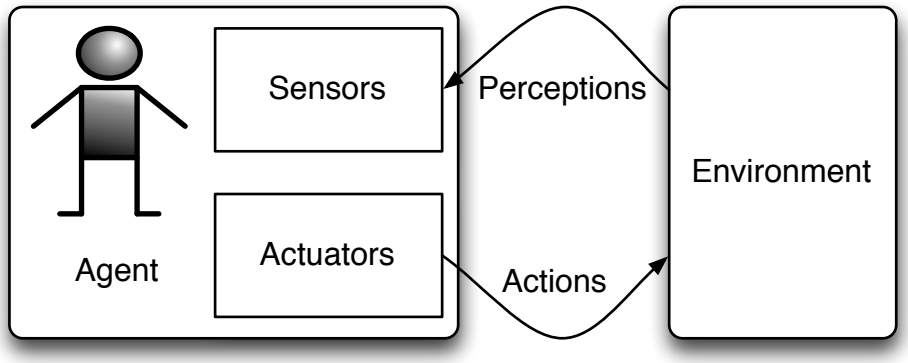
Different definitions of AI



Agent

An agent is something that acts ... in AI:

- acts autonomously
- perceives the environment
- adapts to the environment
- pursues goals...



We need an agent **function** that 'maps' perceptions to actions

So far?

- Learning: when the programmer does not know the agent 'function', we may use learning to improve performance from observations
- We now move to the case in which we know how to formulate solutions (part of the agent function), but we need to find a way to reach a goal among those solutions.
- How?
  By constructing possible solutions and search for one that satisfies the goal

Search in AI

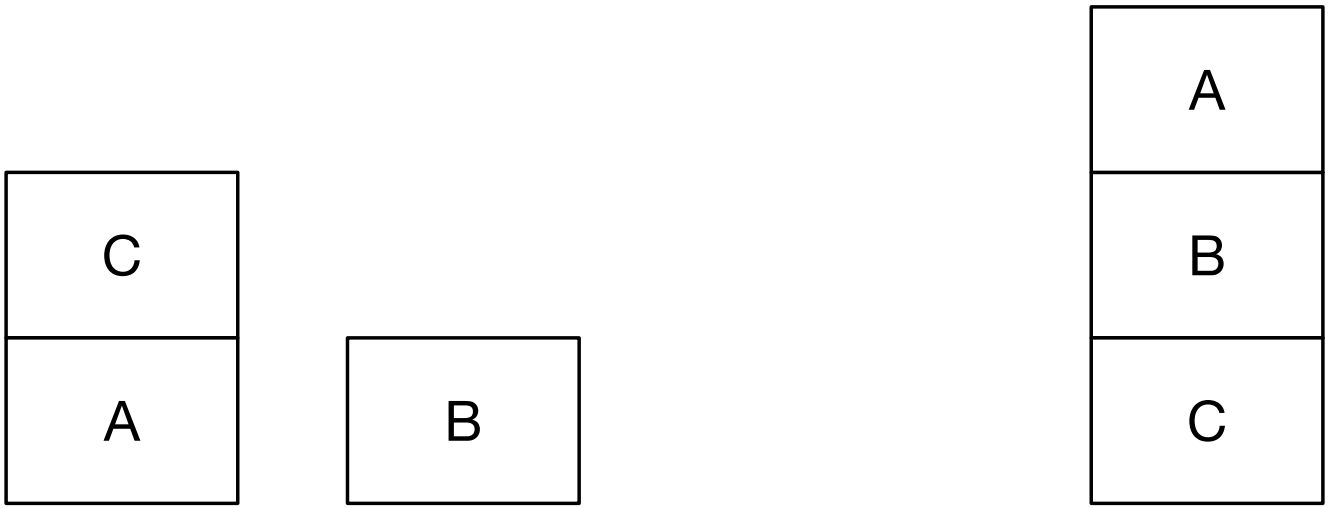
Search is a central topic in Artificial Intelligence, many applications for example...

Example: Route Planning

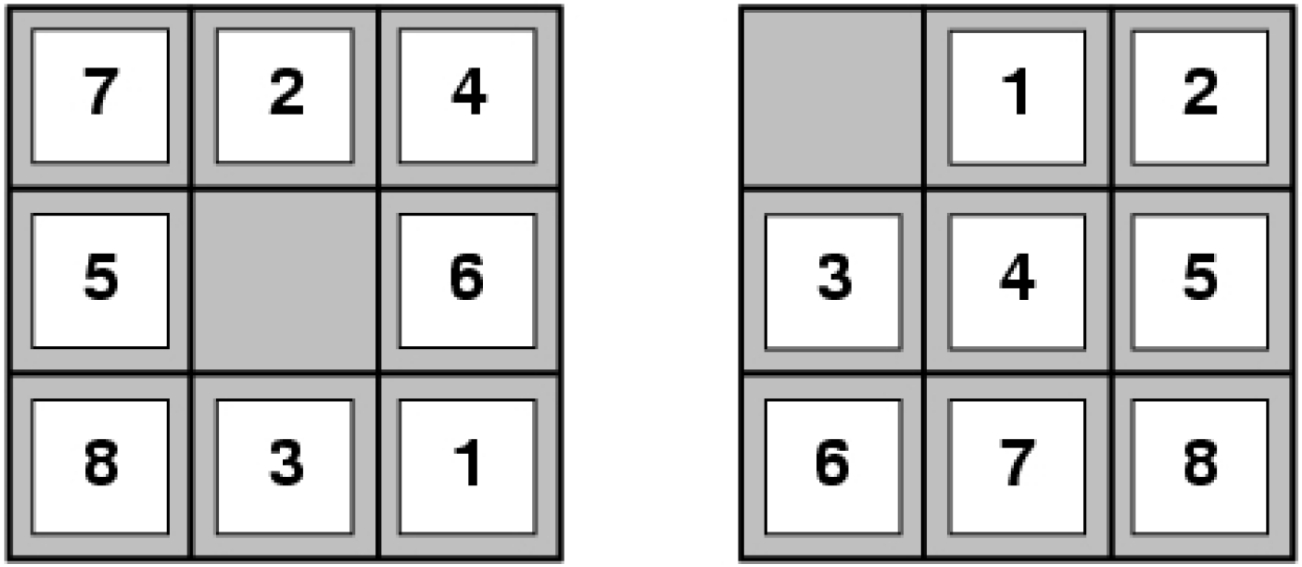
Example: Route Planning

By A. Smith (Taken by A. Smith at RoboCup Rescue 2008 competition.)

Example: Planning

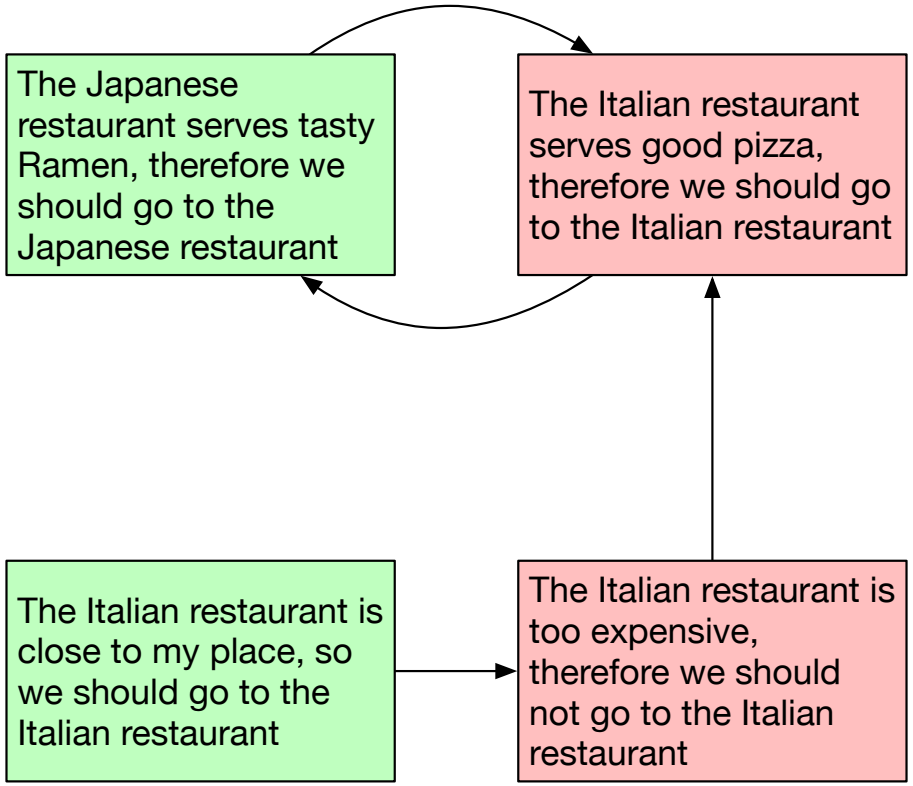


Example: 8-Puzzle



Russel & Norvig

Example: Evaluate arguments



Common components

- An agent is in an **initial situation** and wants to achieve a certain **goal**.
- At any point in time an agent has different simple **actions available**. Executing a particular sequence of such actions may or may not achieve the goal.
- Each action brings the agent from a **state** to another state.
- **Search** is the process of inspecting several such sequences and choosing one that achieves the goal.
- For some applications, each sequence of actions may be associated with a certain **cost**.
- A search problem where the agent aims at reaching the goal but also at doing so at minimal cost is an **optimisation problem**.

Route planning components

- State space: What are the possible states?

Route planning components

- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?

Route planning components

- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?
  - Moves on the map
- Initial State: Where do we start?

Route planning components

- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?
  - Moves on the map
- Initial State: Where do we start?
  - Dundee airport
- Goal: When have we found a solution?

Route planning components

- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?
  - Moves on the map
- Initial State: Where do we start?
  - Dundee airport
- Goal: When have we found a solution?
  - School of Computer Science, St Andrews
- Search: How do we go from initial state to goal?

Route planning components

- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?
  - Moves on the map
- Initial State: Where do we start?
  - Dundee airport
- Goal: When have we found a solution?
  - School of Computer Science, St Andrews
- Search: How do we go from initial state to goal?
  - Navigate through the (possibly huge) space of states (positions on map)
- Cost: How costly is a given move?

Route planning components

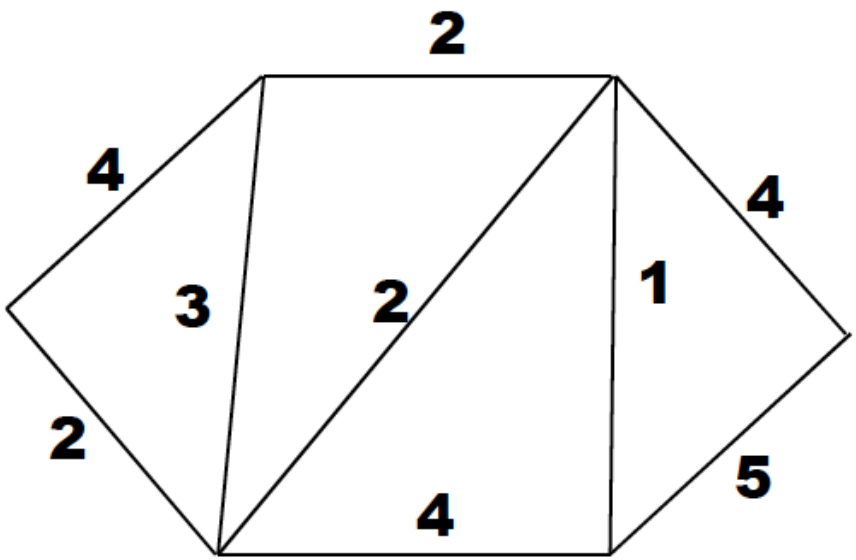
- State space: What are the possible states?
  - Position on the map
- Actions: What are legal moves between states?
  - Moves on the map
- Initial State: Where do we start?
  - Dundee airport
- Goal: When have we found a solution?
  - School of Computer Science, St Andrews
- Search: How do we go from initial state to goal?
  - Navigate through the (possibly huge) space of states (positions on map)
- Cost: How costly is a given move?
  - Distance, fuel cost, etc...

Another Example: Travel Salesperson 1

- Travel around a set of cities visiting each city exactly once and return home
- Traditionally, problem is to minimise total distance travelled
- Can also be cast as a decision problem: does there exist a tour with distance $\leq$ limit

Another Example: Travel Salesperson 2

- Problem: graph with a cost for each edge
- Solution: tour visiting all nodes exactly once, returning to base meeting some cost limit (or reaching minimum cost)
  - e.g. minimum cost is 21

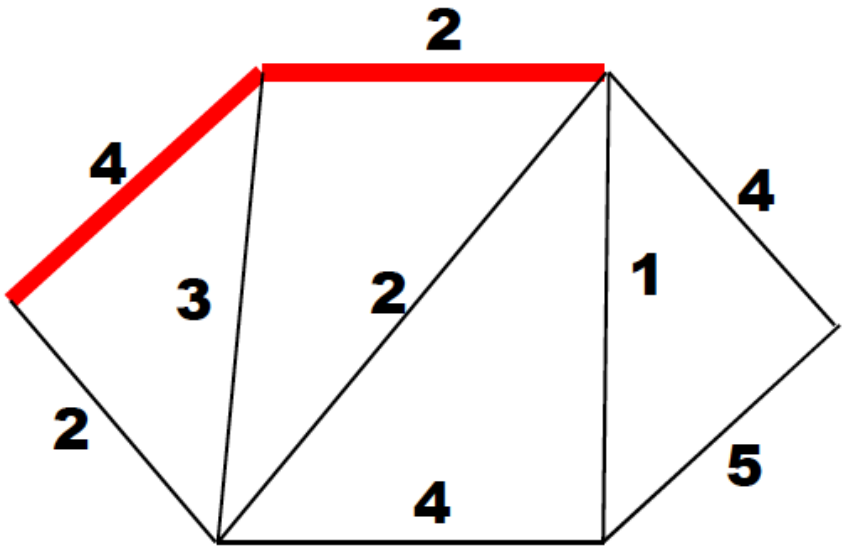


Another Example: Travel Salesperson 3

- Easy to check that tour costs no more than limit
- Complete search might need to check every possible tour
  - If a poor algorithm is used
  - We can do much better than checking every tour

Another Example: Travel Salesperson 4

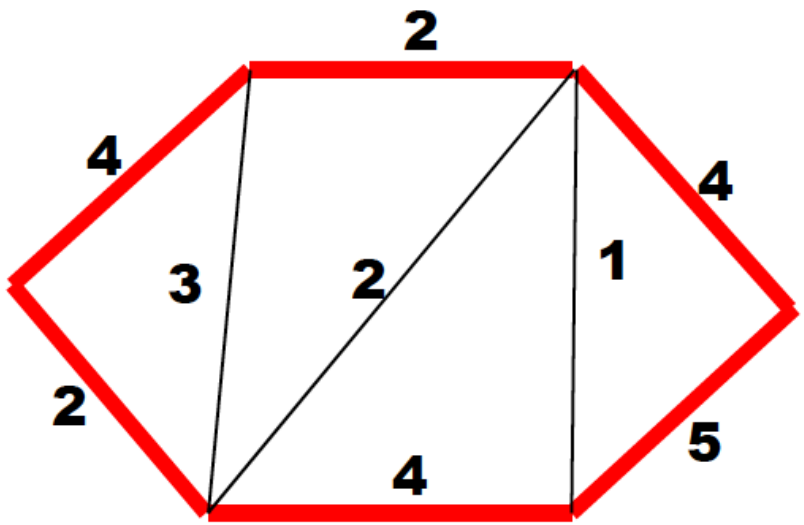
- A search state is a partial tour or a complete tour
- A partial tour:



- N.B. This is just one way of formulating TSP for search

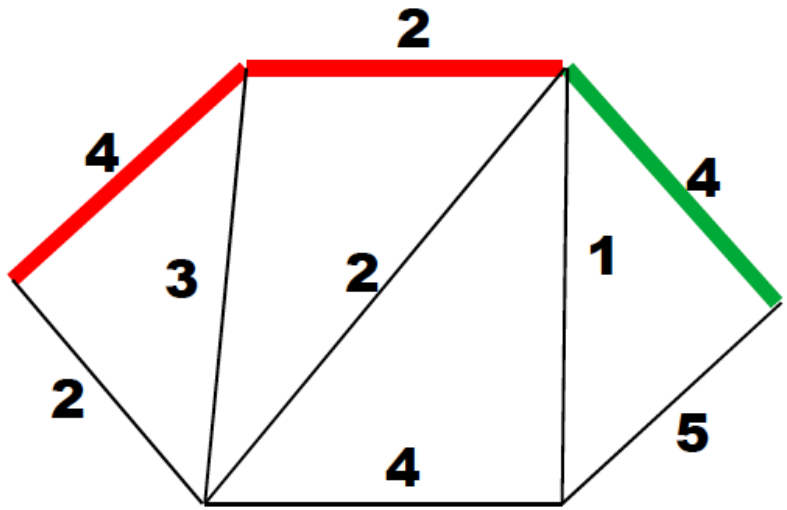
Another Example: Travel Salesperson 5

- A solution is a complete tour (within the distance limit)
  - A special type of search state
- A complete tour:



Another Example: Travel Salesperson 6

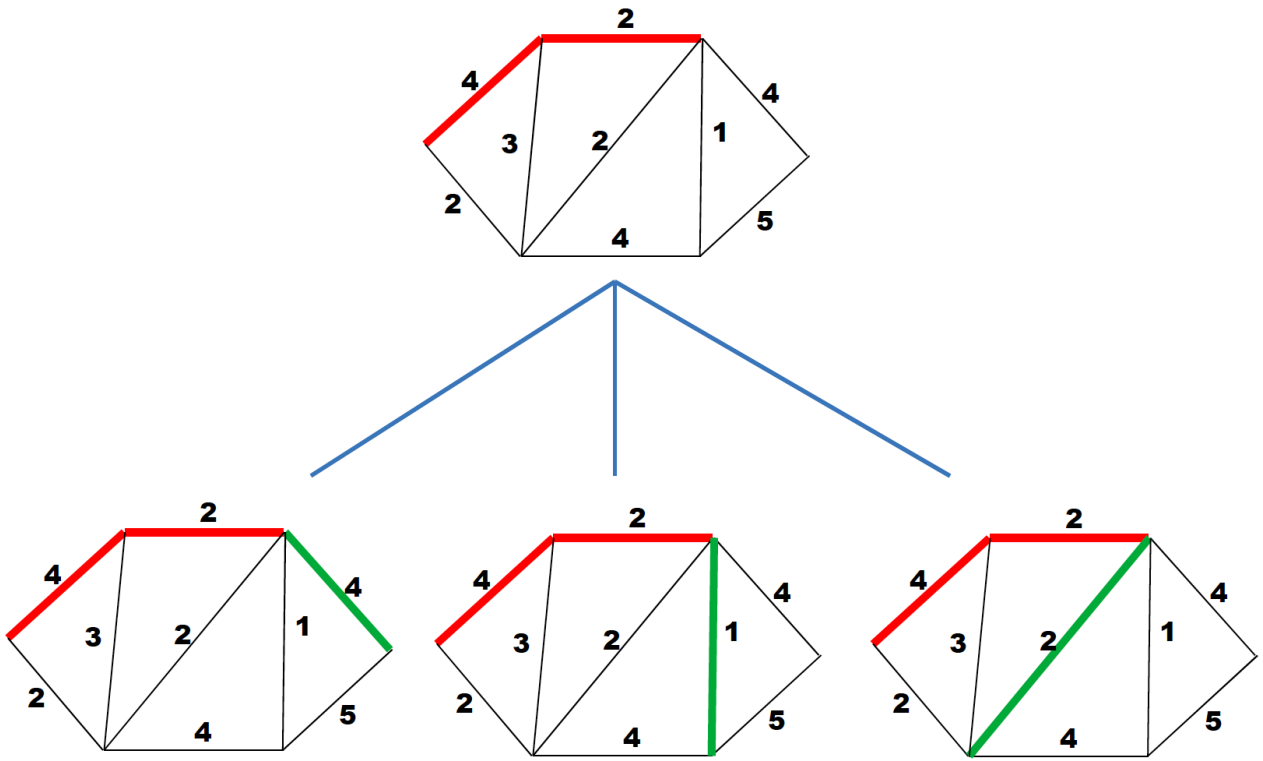
- We can construct a solution (if one exists) by repeatedly extending the empty tour?
- A partial tour (red) extended (in green):



Another Example: Travel Salesperson 7

- However, it might not be clear how to extend a search state
- There might be more than one option
- We need to explore a **search tree**
- Search trees are an abstraction, we never store them in memory

Another Example: Travel Salesperson 8



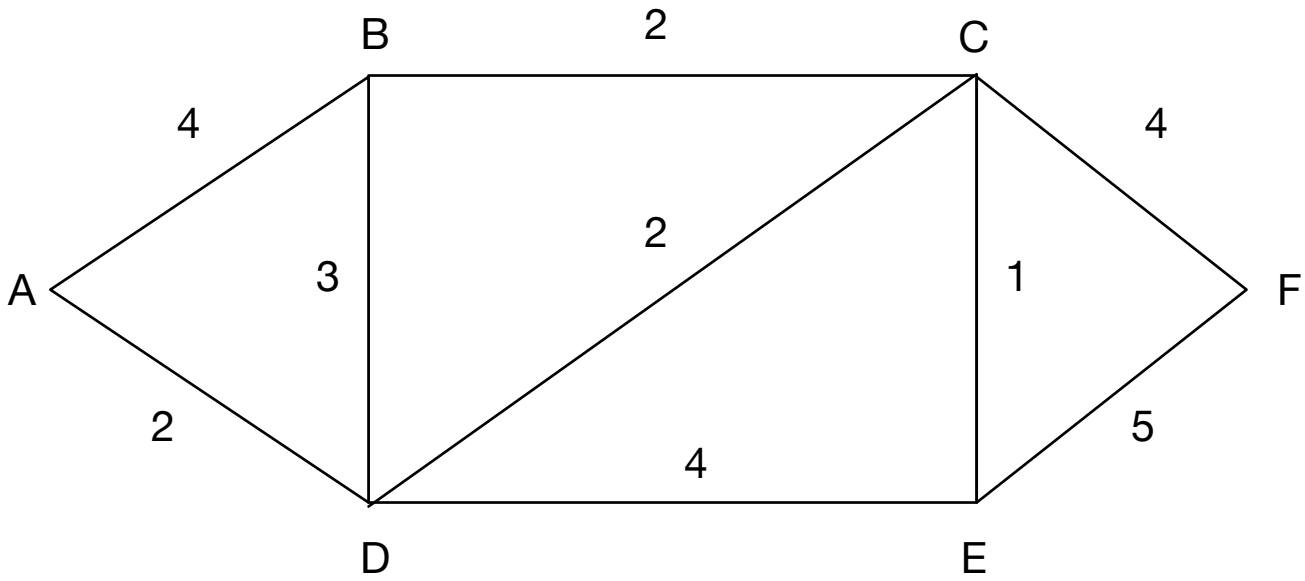
Problem Formulation

What to define:

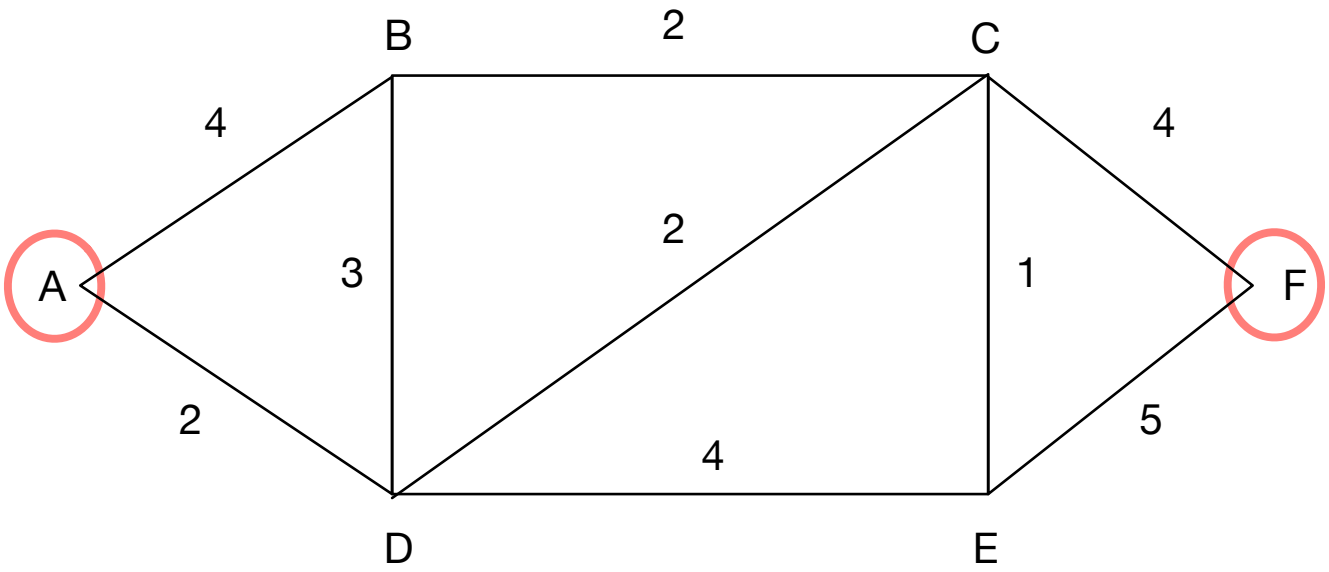
- State space
- Actions: moves between states
- Initial State
- Goal
- Search (successor function)
- Cost

Note: The type of problem formulation can have a big influence on the difficulty of finding a solution.

Let's focus on a slightly different problem...



Let's focus on a slightly different problem... find the route from A to F

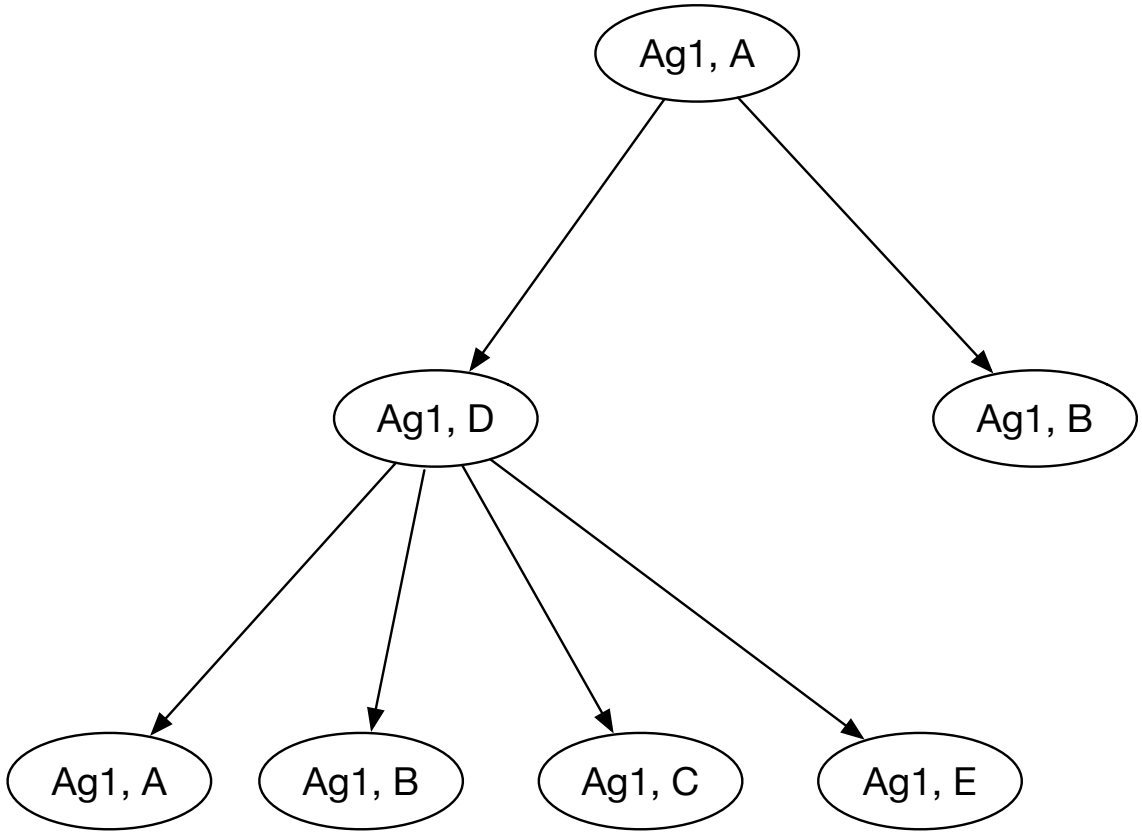


Problem formulation

- State space: agent *Ag1* in a location *locX* (eg. $\langle Ag1, C \rangle$)
- Initial State: $\langle Ag1, A \rangle$
- Successor function: Ag1 moves from *locA* to an adjacent *locB*
  Note: States may be redundant or illegal
- Goal State: $\langle Ag1, F \rangle$
- Cost: **step cost** cost of one move from *locA* to *locB*
  path cost sum of the step costs so far

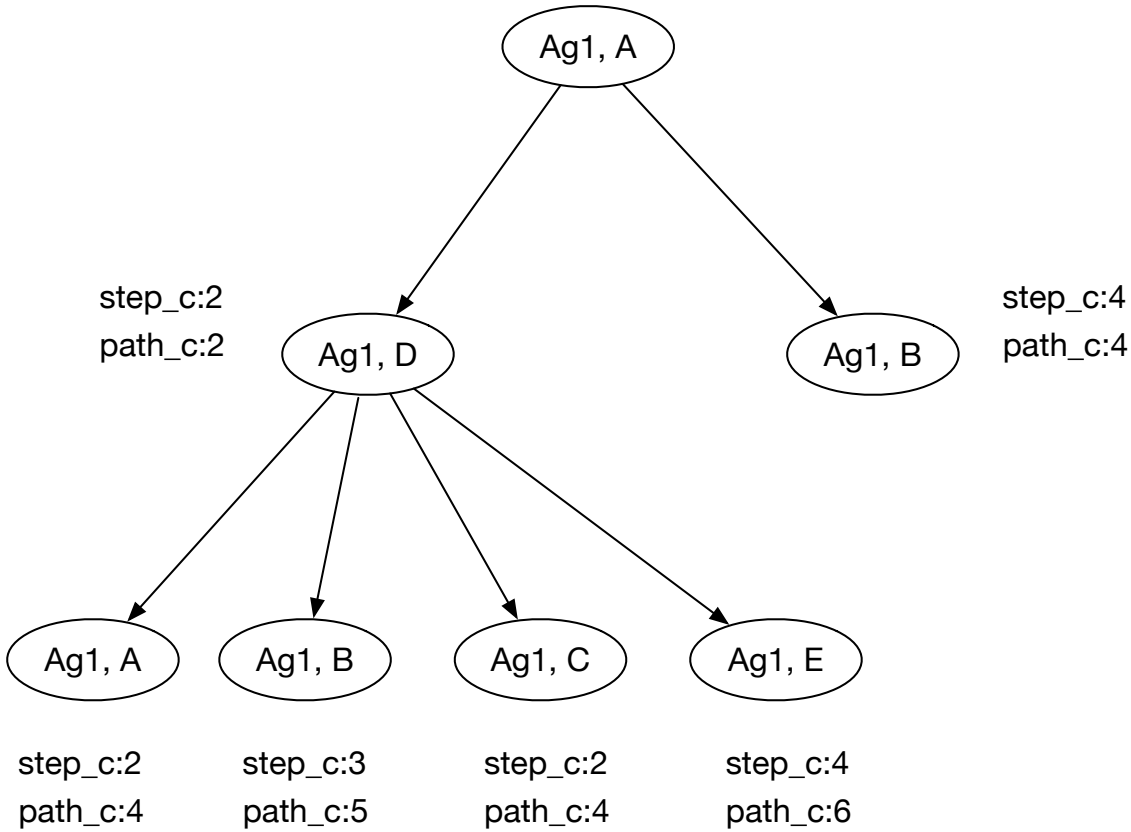
General search

From the initial state, produce all successive states step by step $\rightarrow$ search tree.



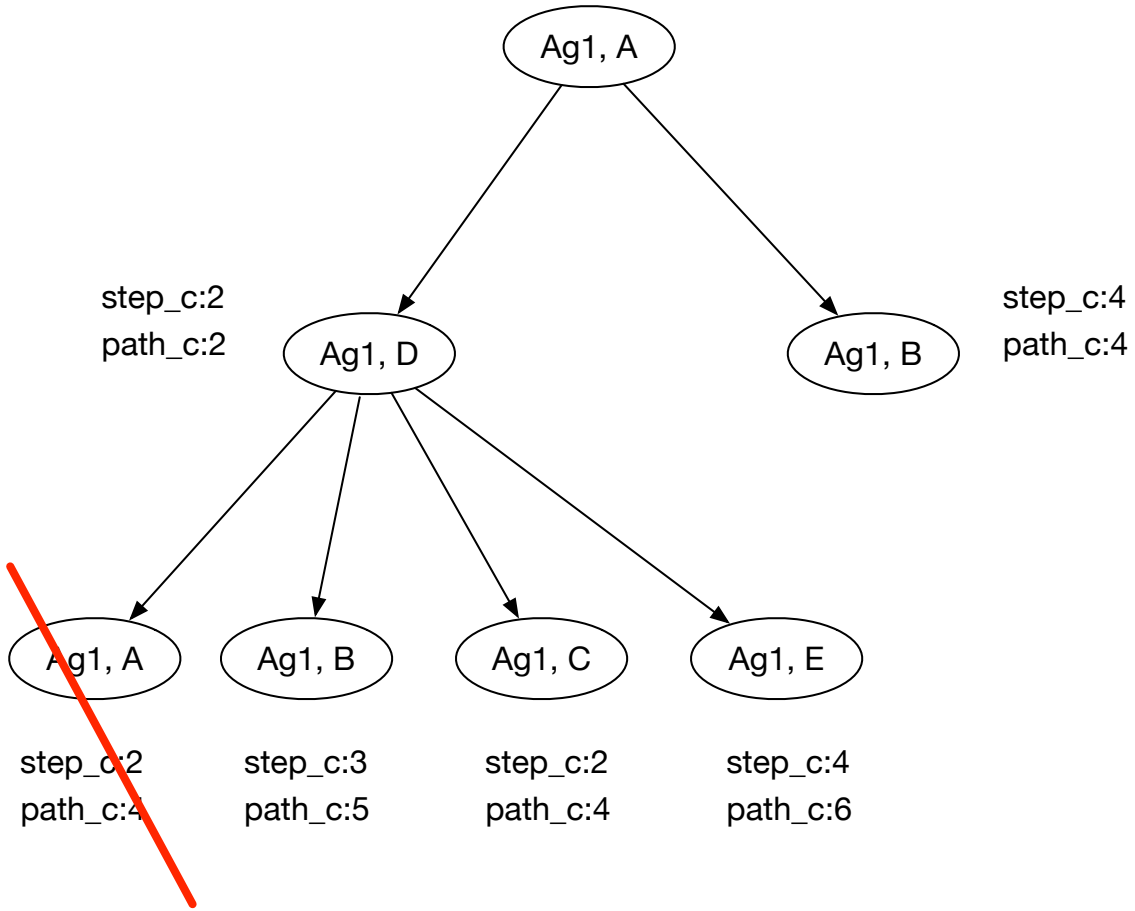
Cost

From the initial state, produce all successive states step by step → search tree.



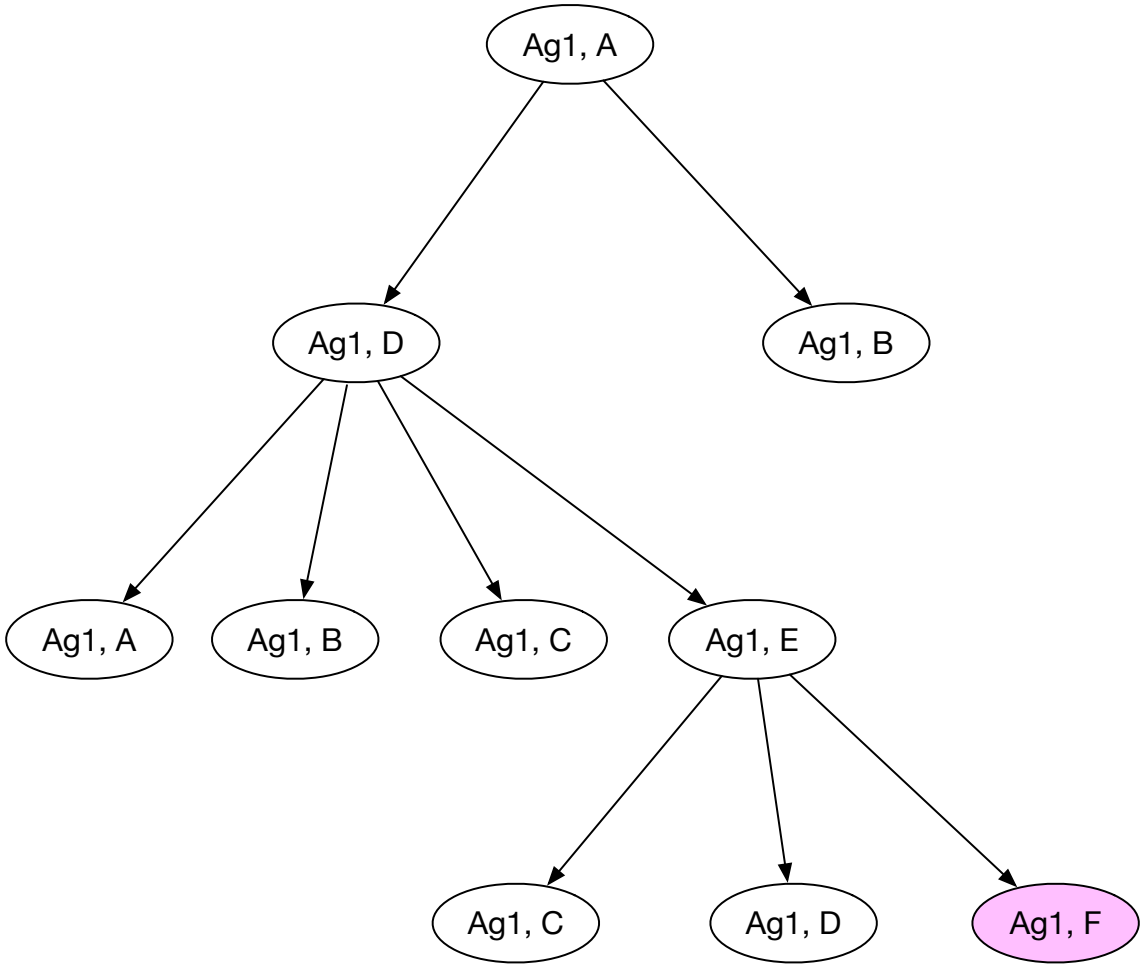
Redundant state

From the initial state, produce all successive states step by step → search tree.



Finding the goal

From the initial state, produce all successive states step by step → search tree.



Implementing Search - Data Structures

Data structure for nodes in the search tree:

- State: State in the state space
- Node: Containing a state, pointer to predecessor, action, depth, and path cost
- Depth: number of steps along the path from the initial state
- Path Cost: Cost of the path from the initial state to the node
- Frontier: Memory for storing expanded nodes. Eg., a stack or a queue

Implementing Search - Functions

General functions to implement:

- Make-Node(Node node, State state): Creates a node from a state
- Goal-Test(State state,State goal): Returns true if state is a goal state
- Successor-Fn(State node, Problem problem): Implements the successor function, i.e. expands a set of new nodes given all actions applicable in the state
- Cost(State state1,State state2): Returns the cost for executing action in state
- Insert(Node node, Frontier frontier): Inserts a new node into the frontier
- Remove(int index, Frontier frontier): Returns the first node from the frontier

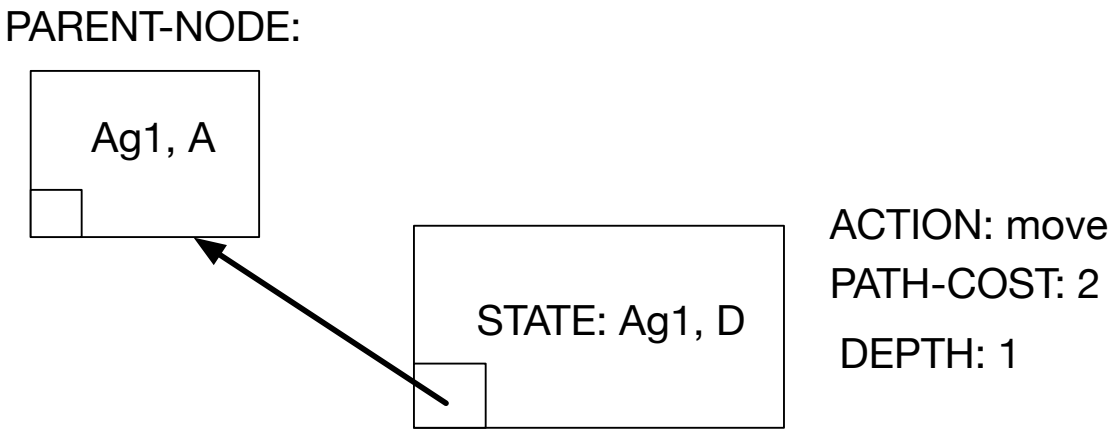
General Search Algorithm

function MAKE-NODE(*node*,*state*) **returns** a node

$n \leftarrow$ a new Node
STATE[n] $\leftarrow$ state
PARENT-NODE[n] $\leftarrow$ node
ACTION[n] $\leftarrow$ move from STATE[*node*] to *state*
PATH-COST[n] $\leftarrow$ PATH-COST[*node*] + COST(STATE[*node*],state)
DEPTH[n] $\leftarrow$ DEPTH[*node*] +1
return n

end

Node



General Search Algorithm

function TREE-SEARCH(*problem,frontier*) **returns** a solution, or failure

initial_node ← MAKE-NODE(*null,initial_state*)

frontier ← INSERT(*initial_node*, *frontier*)

loop do
if *frontier* is empty **return** failure
nd ← REMOVE(*index*, *frontier*)
if GOAL-TEST(STATE[*nd*], *goal*)
return *nd*
else
frontier ← INSERT-ALL (EXPAND(*nd*, *problem*, *frontier*))
end loop

end

General Search Algorithm

function EXPAND(*node, problem, frontier*) **returns** a set of nodes

next_states $\leftarrow$ SUCCESSOR-FN(STATE[*node*],*problem*)

successors $\leftarrow$ empty set

    for each *state* **in** *next_states*
    nd $\leftarrow$ MAKE-NODE(*node*,*state*)
    add *nd* to *successors*
    end for

return *successors*

end

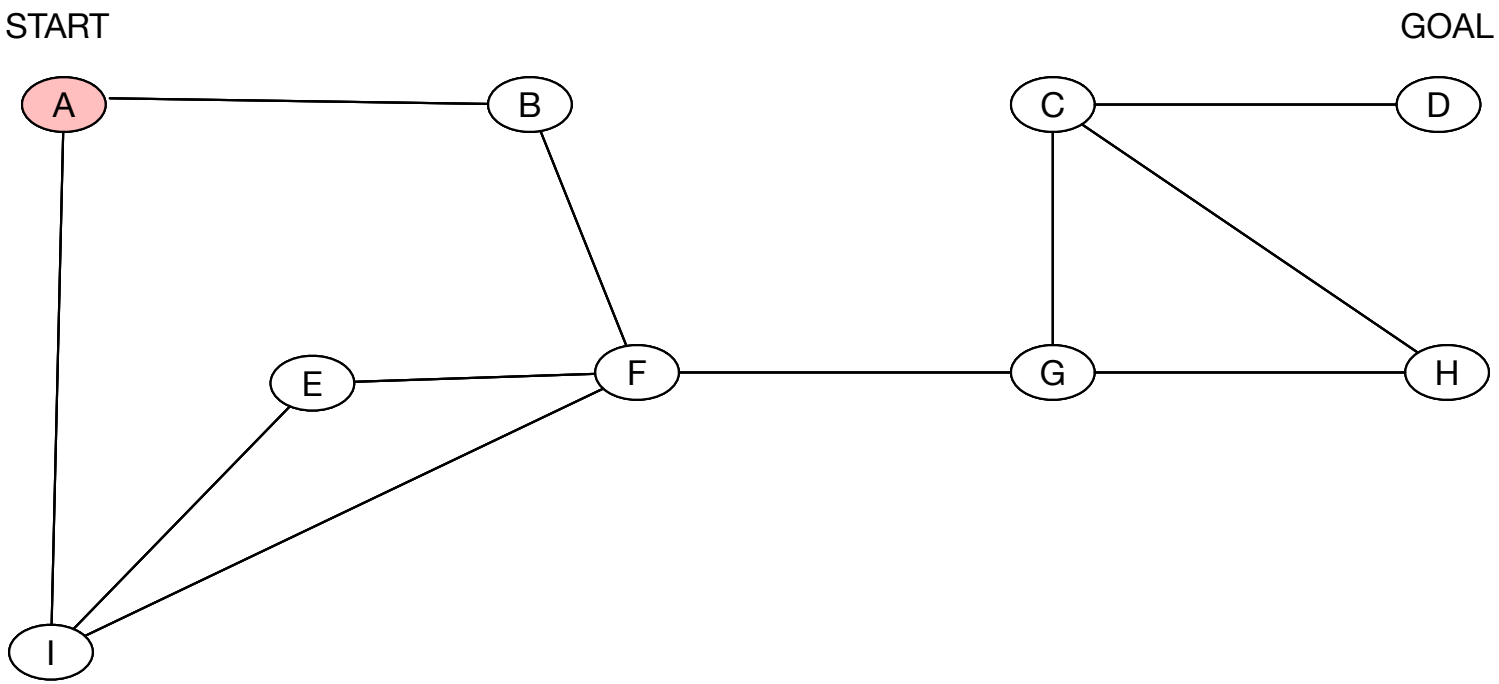
Search Strategies

- Uninformed or blind searches:
  - No information on the length or cost of a path to the solution
- Informed search:
  - Uses information on the length or cost of a path to choose the next step (heuristics)

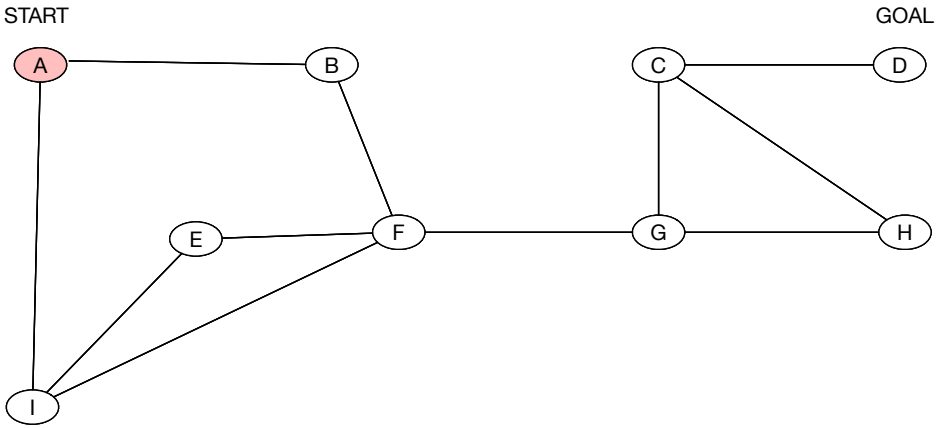
Search Strategies

- Uninformed or blind searches:
  - breadth-first search
  - depth-first search
  - uniform-cost search
  - iterative deepening
- Informed search (strategy+heuristics)
  - Greedy Best first
  - A*

Breadth-first search

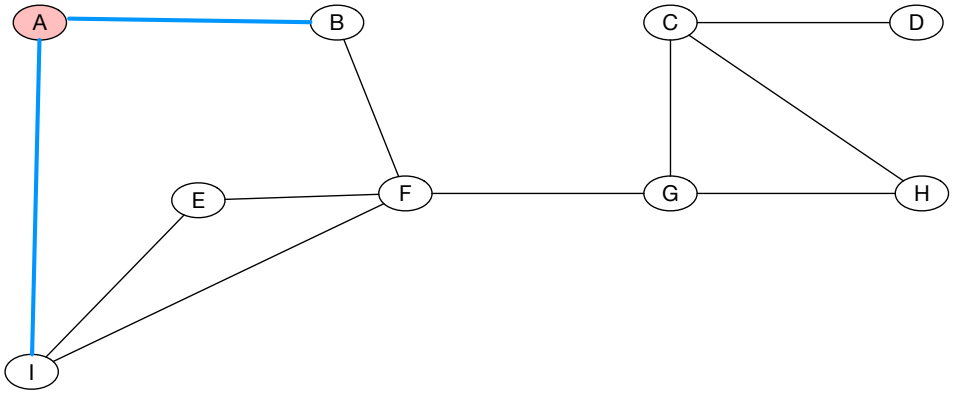


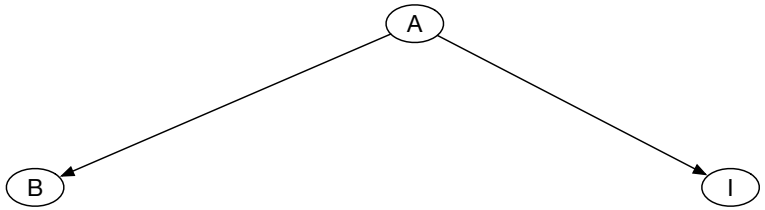
Breadth-first search



- Current node:
- Frontier: {A}
- Explored: { }

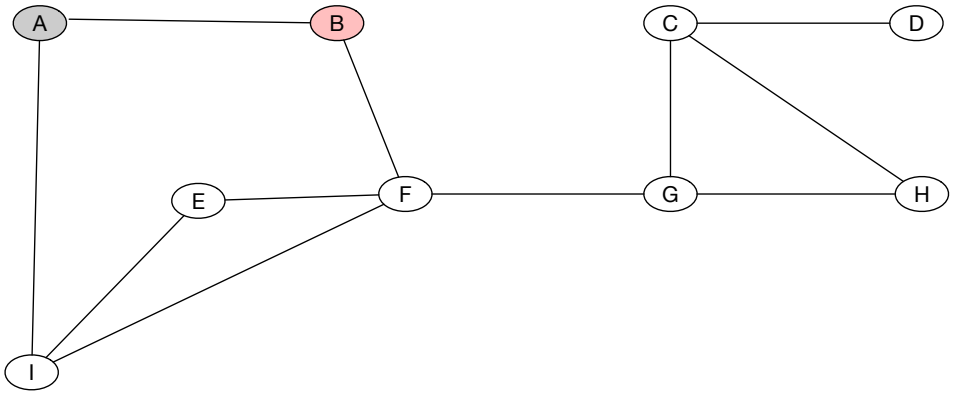
Breadth-first search

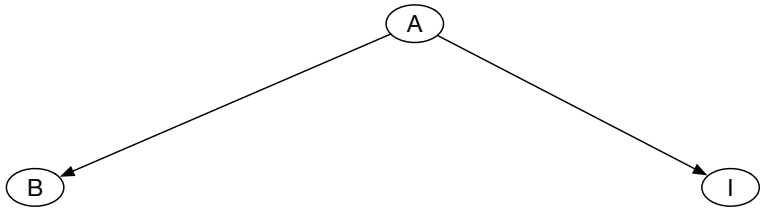




- Current node: A
- Frontier: {B,I}
- Explored: { A }

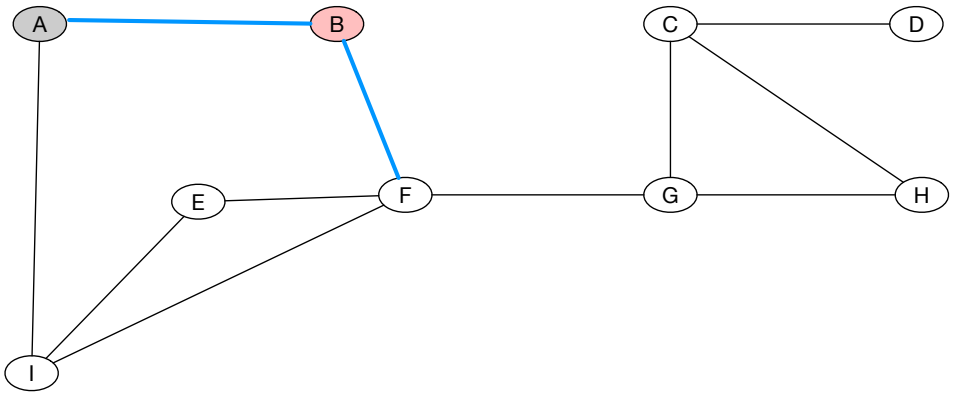
Breadth-first search

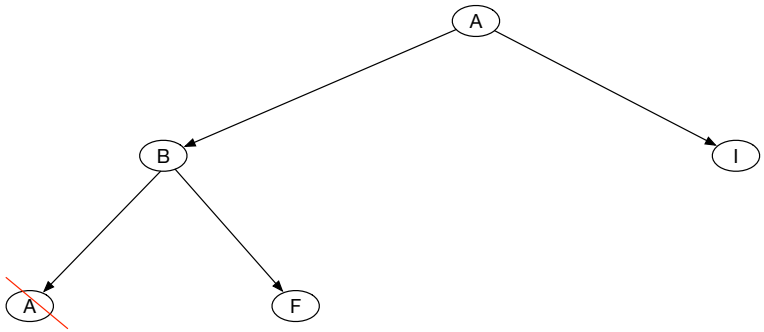




- Current node: B
- Frontier: {I}
- Explored: { A }

Breadth-first search

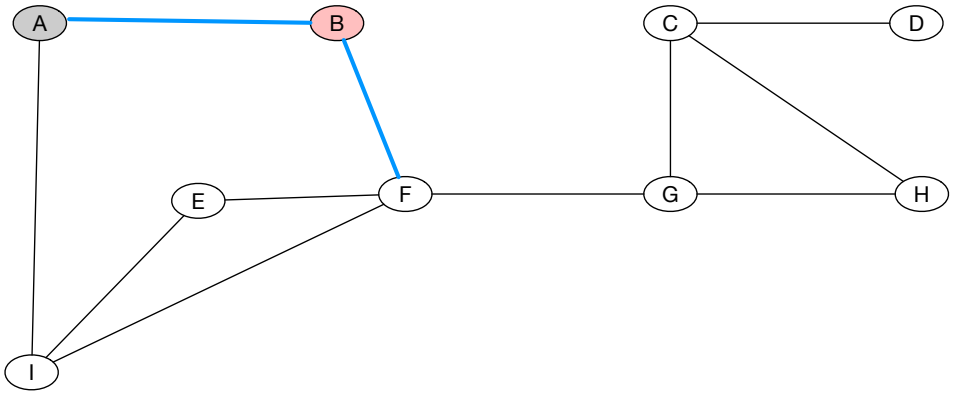


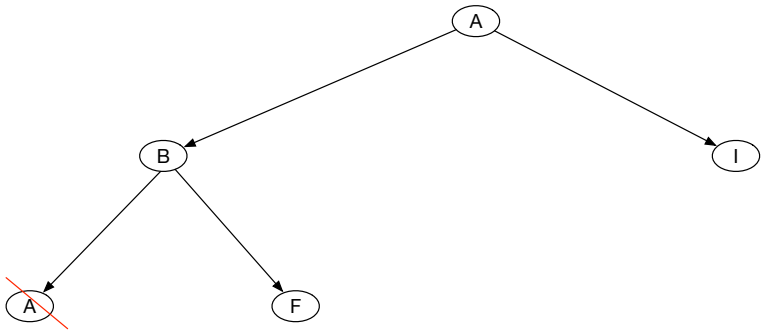


- Current node: B
- Frontier: {I,A,F}
- Explored: { A ,B }

But A has been explored already

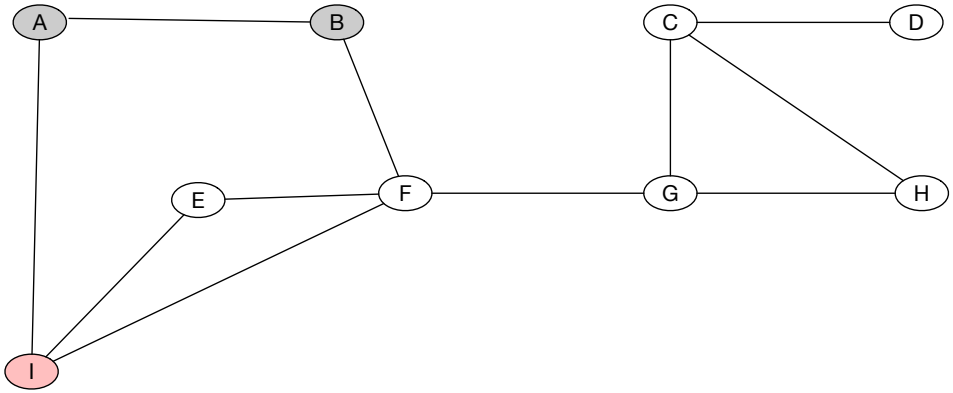
Breadth-first search

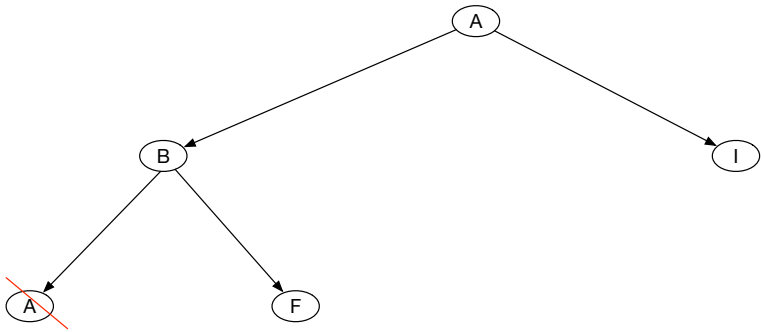




- Current node: B
- Frontier: {I,F}
- Explored: { A ,B }

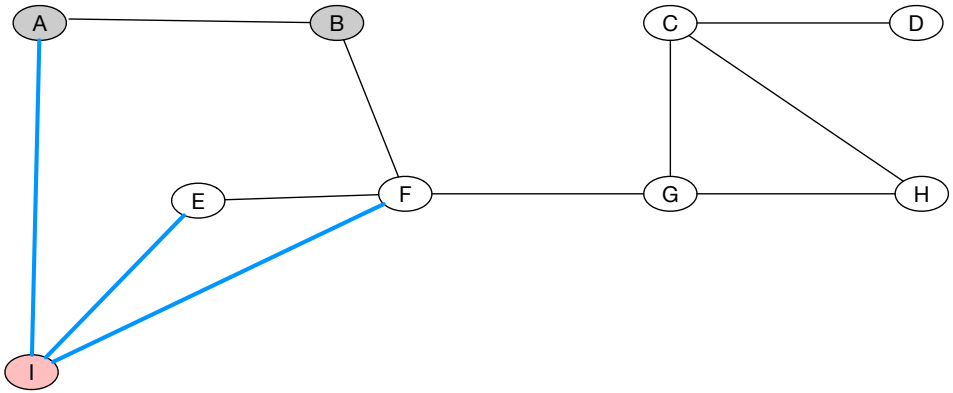
Breadth-first search

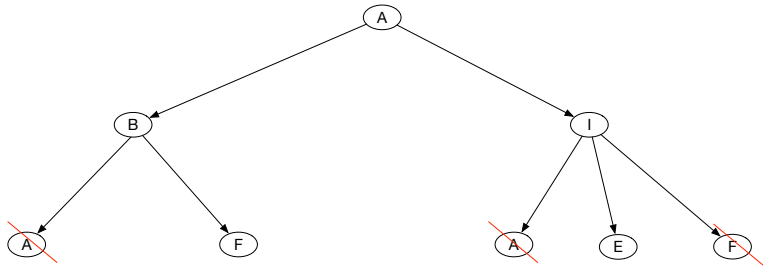




- Current node: I
- Frontier: {F}
- Explored: { A ,B }

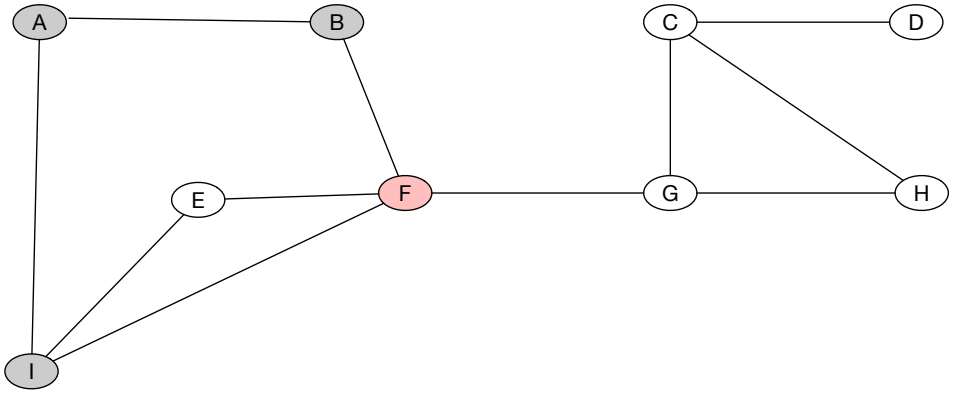
Breadth-first search

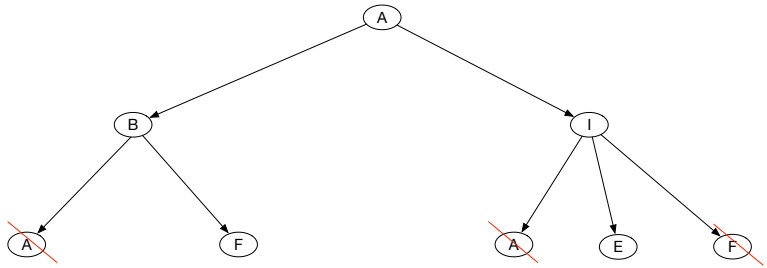




- Current node: I
- Frontier: {F,E}
- Explored: { A ,B ,I }

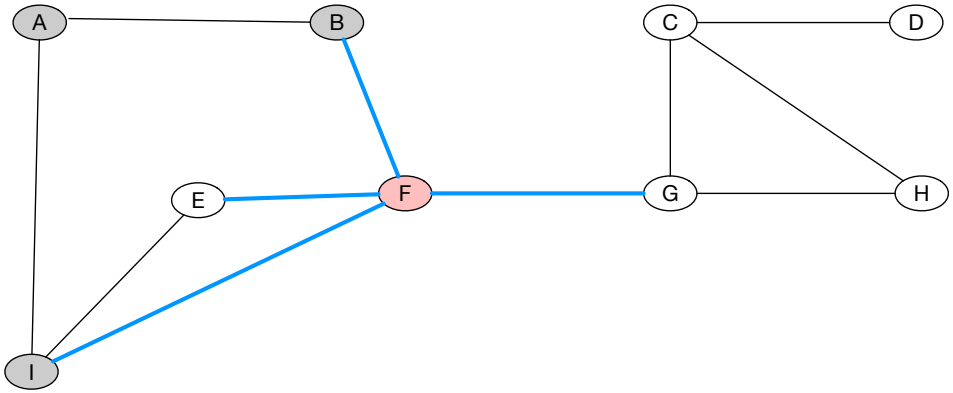
Breadth-first search

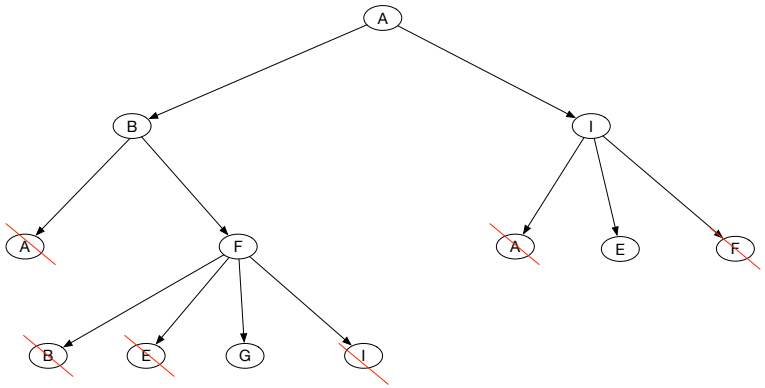




- Current node: F
- Frontier: {E}
- Explored: { A ,B ,I }

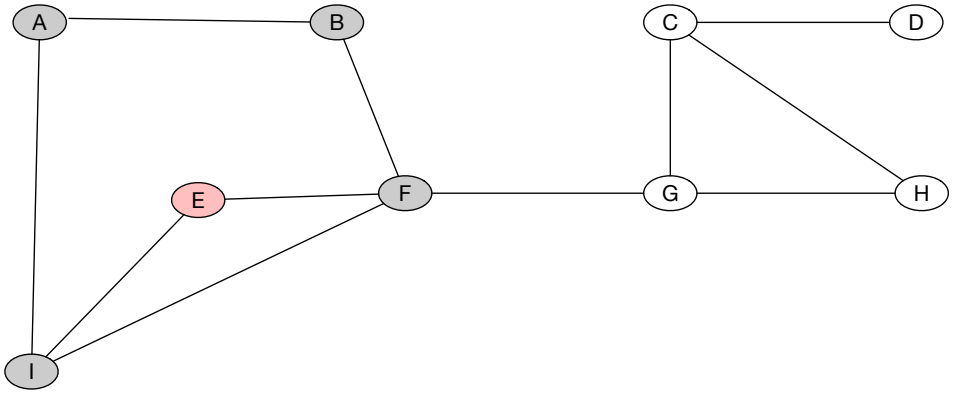
Breadth-first search

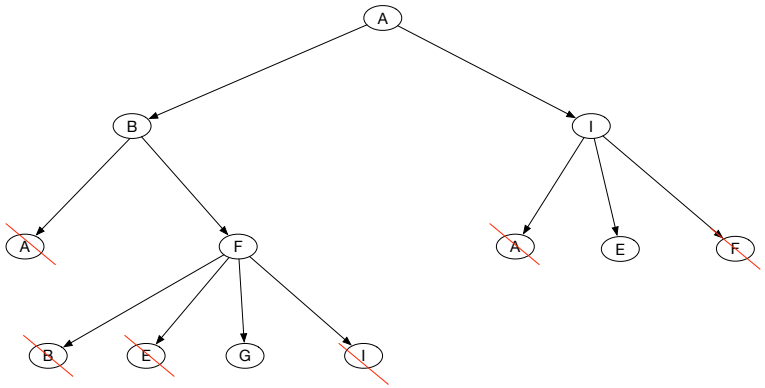




- Current node: F
- Frontier: {E,G}
- Explored: { A ,B ,I ,F }

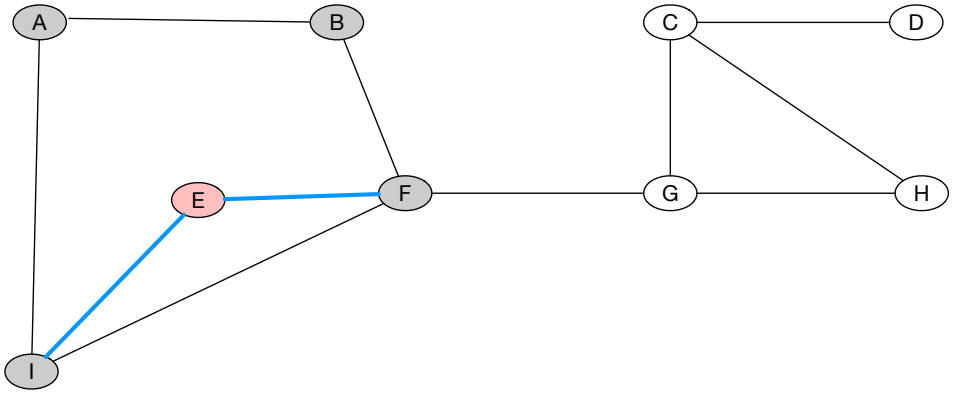
Breadth-first search

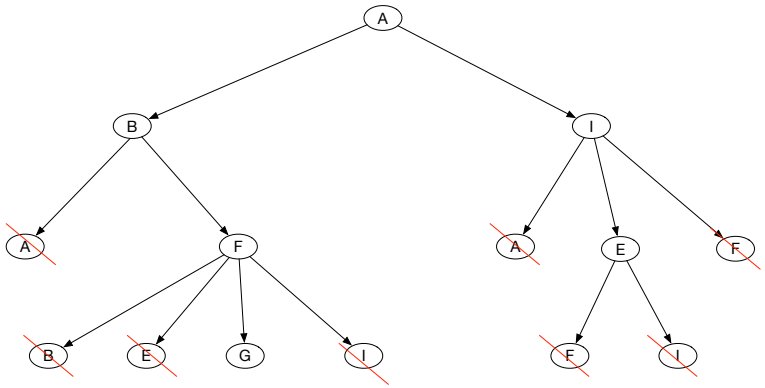




- Current node: E
- Frontier: {G}
- Explored: { A ,B ,I ,F }

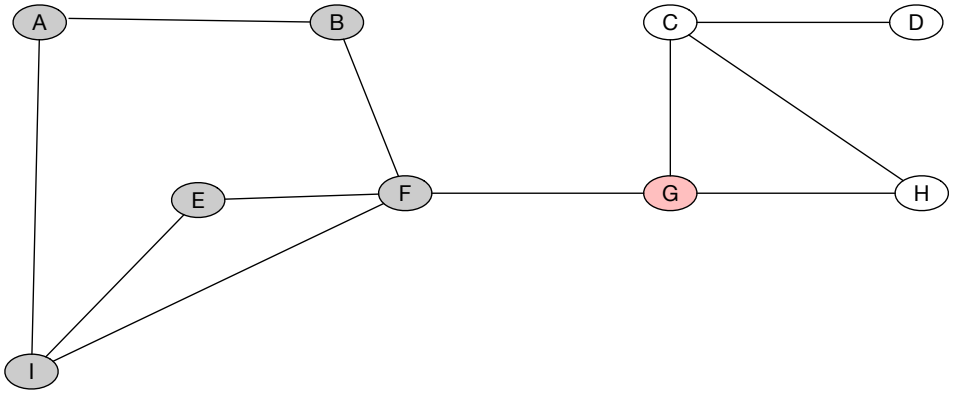
Breadth-first search

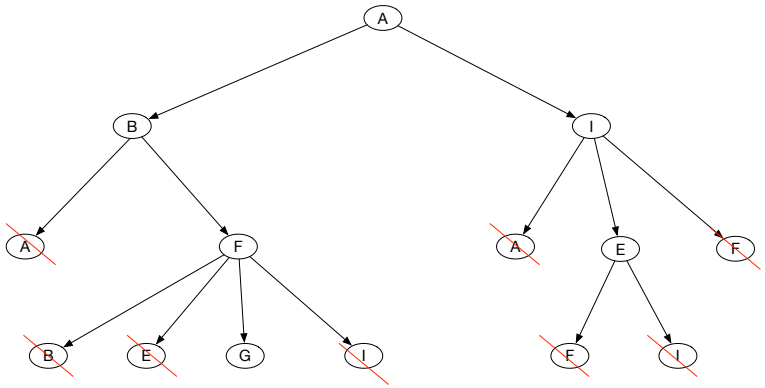




- Current node: E
- Frontier: {G}
- Explored: { A ,B ,I ,F ,E }

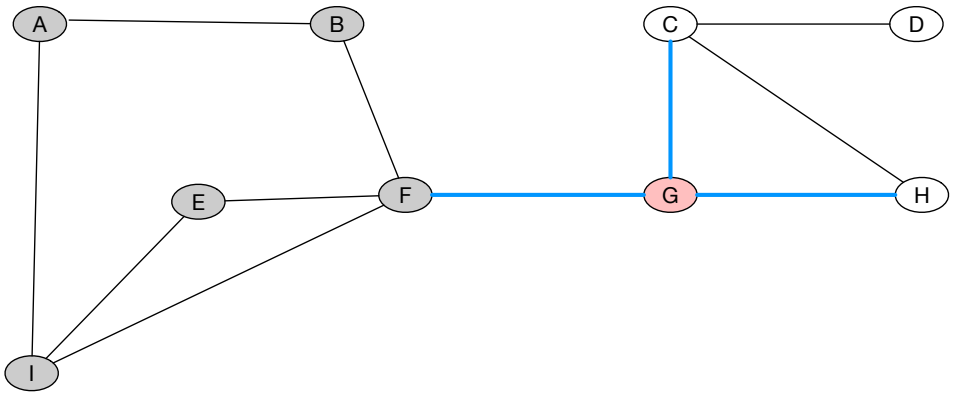
Breadth-first search

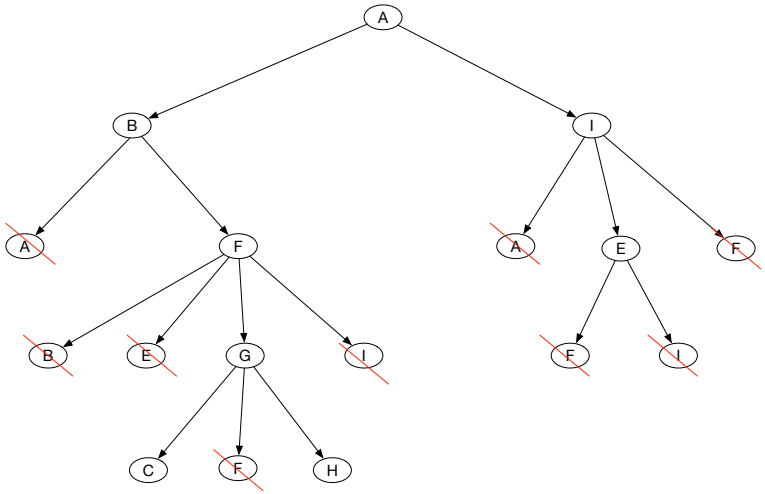




- Current node: G
- Frontier: {}
- Explored: { A ,B ,I ,F ,E }

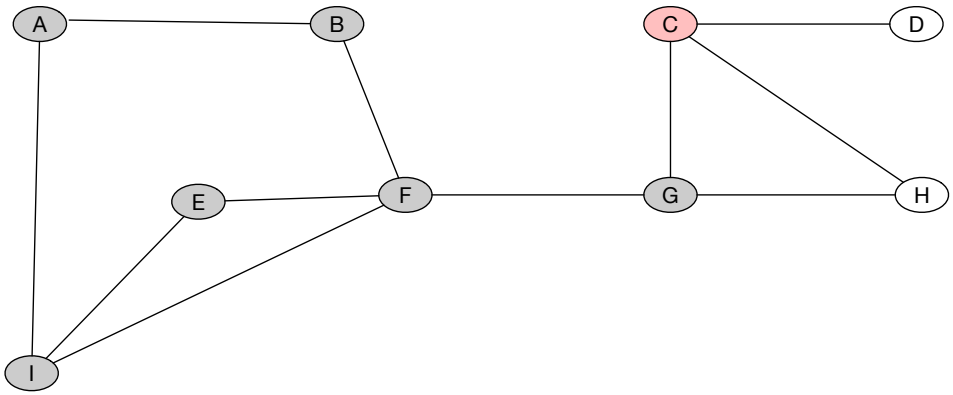
Breadth-first search

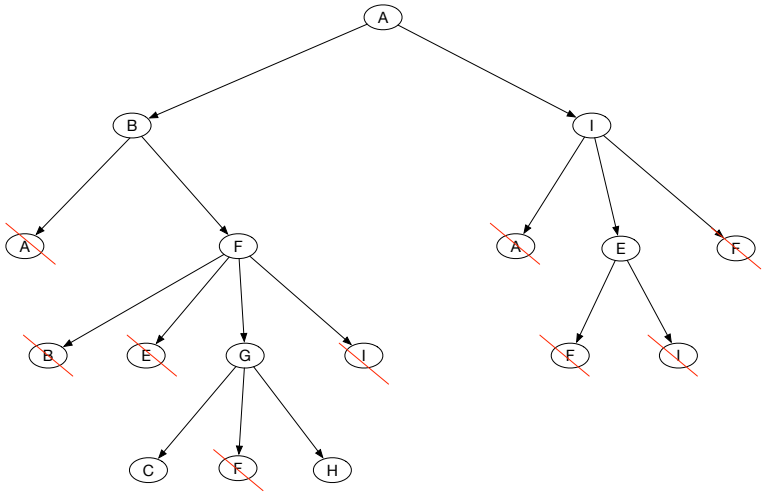




- Current node: G
- Frontier: {C,H}
- Explored: { A ,B ,I ,F ,E ,G }

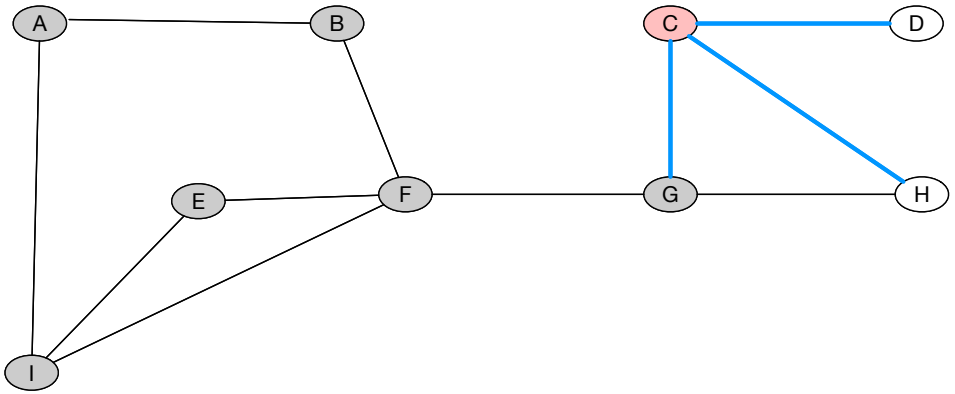
Breadth-first search

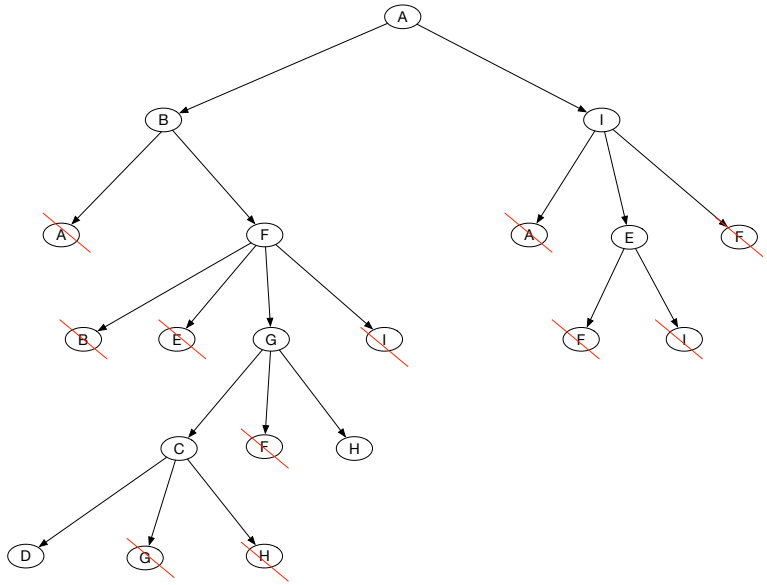




- Current node: C
- Frontier: {H}
- Explored: { A ,B ,I ,F ,E ,G }

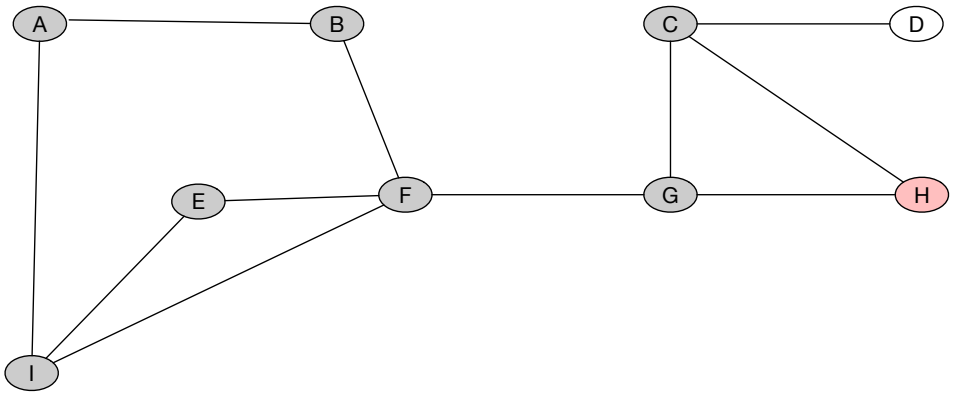
Breadth-first search

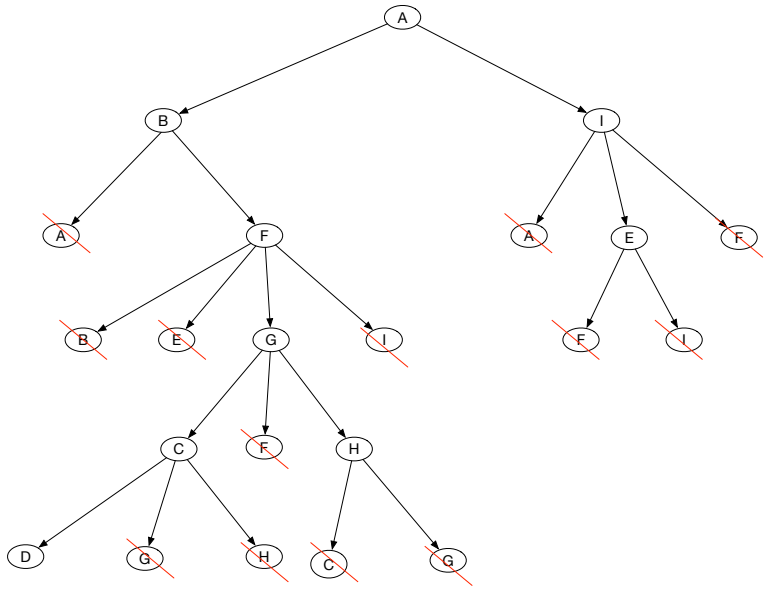




- Current node: C
- Frontier: {H,D}
- Explored: { A ,B ,I ,F ,E ,G ,C }

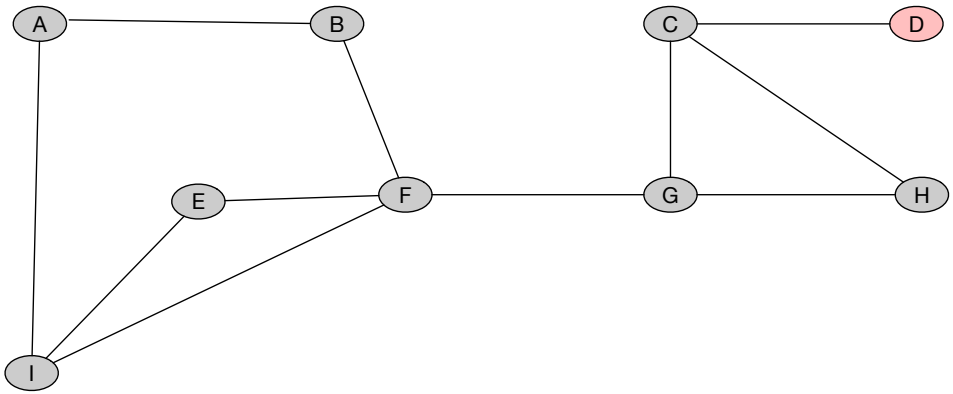
Breadth-first search

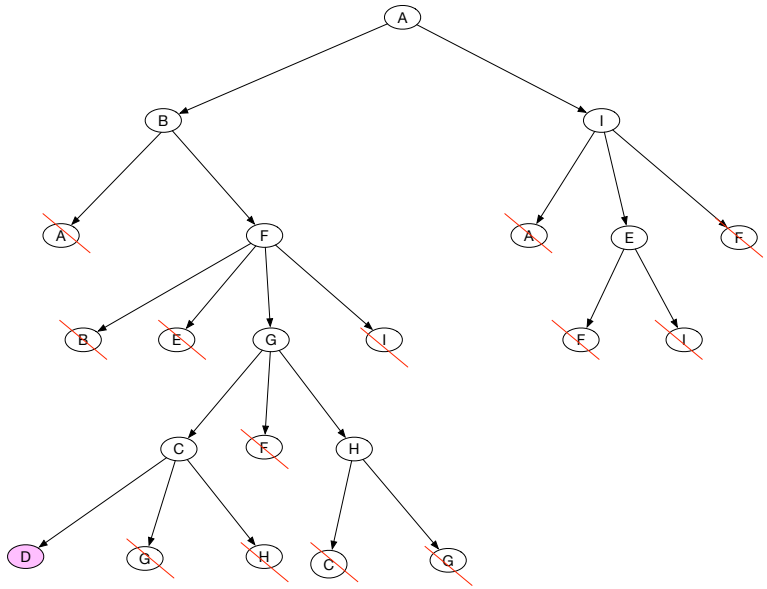




- Current node: H
- Frontier: {D}
- Explored: { A ,B ,I ,F ,E ,G ,C ,H }

Breadth-first search



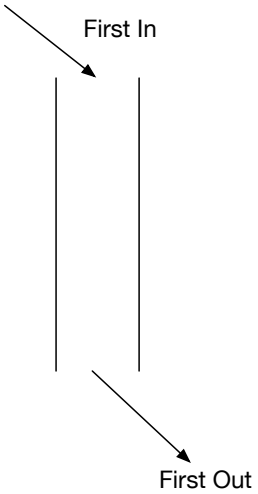


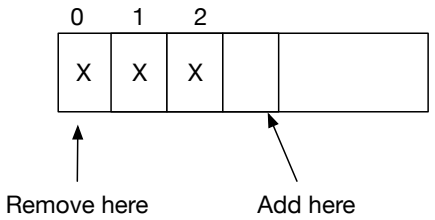
- Current node: D
- Frontier: {}
- Explored: { A ,B ,I ,F ,E ,G ,C ,H ,D }

Breadth-first search

Expand all nodes at the same level & use:

- First-in-first-out (FIFO) $\rightarrow$ Queue structure for Frontier





- to avoid loops: check that explored nodes and frontier do not include the new node
- for this problem: SUCCESSOR-FN $\rightarrow$ connected node in alphabetical order

Search Algorithm Example

function TREE-SEARCH(*problem*,*frontier*) **returns** a solution, or failure

initial_node ← MAKE-NODE(*null*,*initial_state*)

frontier ← INSERT-END(*initial_node*, *frontier*)

explored ← empty set

loop do

if *frontier* is empty **return** failure

nd ← REMOVE-FRONT(*frontier*)

Add *nd* to *explored*

if GOAL-TEST(STATE[*nd*], *goal*)

return *nd*

else

frontier ← INSERT-ALL (EXPAND(*nd*, *problem*, *frontier*, *explored*))

end loop

end

General Search Algorithm

function EXPAND(*node*, *problem*, *frontier*, *explored*) **returns** a set of nodes

next_states ← SUCCESSOR-FN(STATE[*node*],*problem*)

successors ← empty set

for each *state* **in** *next_states*
nd ← MAKE-NODE(*node*,*state*)
if nd is not in *explored* or *frontier* then
add *nd* to *successors* in alphabetical order
end for

return *successors*

end

Depth-first search

For next time try and compare the two approaches

Summary

Search in AI:

- Search problems
- Search components
- Search trees
- BFS implementation

Next time

No class tomorrow 10th due to DLS

Assignment 2 and informed search

Reading Material

- S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Pearson Education, 2010 [Chapter 3]