# Solving problems by search

We now look at how an agent might achieve its goals using *search*.

**Aims:**

- to show how problem-solving can be modelled as the process of searching for a sequence of actions that achieves a goal;
- to introduce some basic algorithms for conducting the necessary search for a sequence of actions.

**Reading:** Russell and Norvig, chapter 3.

# Problem solving by basic search

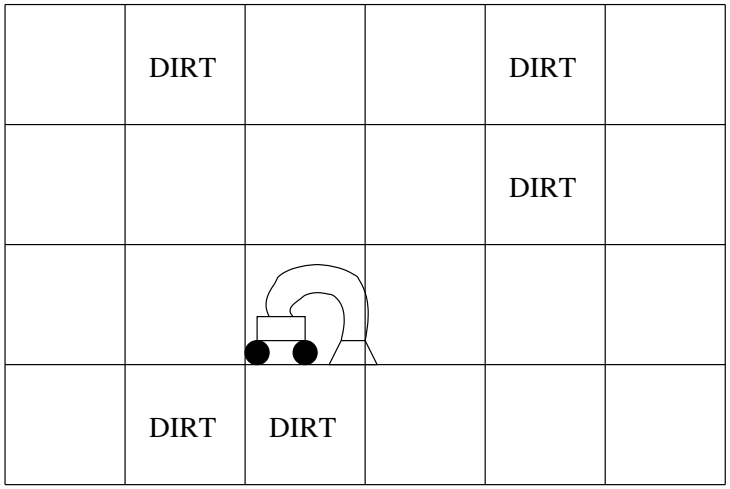
**As usual:** an *agent* exists within an *environment* and must *act* within this environment to achieve some desirable *goal*. It has some means of knowing the *state* of its environment.

**For example:**

- the agent is a robotic vacuum cleaner;
- the environment is a rectangular room with no obstacles, containing the cleaner and some dirt;
- the available actions are movement in four directions, switch on sucker, and switch off sucker;
- the cleaner can sense the presence or otherwise of dirt and knows it's own location;
- the goal is to have no dirt in the room.

# Problem solving by basic search

The situation looks something like this:




Even this simple description hides a number of ambiguities and subtleties.

# Problem solving by basic search

The example given admits a simple solution strategy that is applicable to many simple problems in AI.

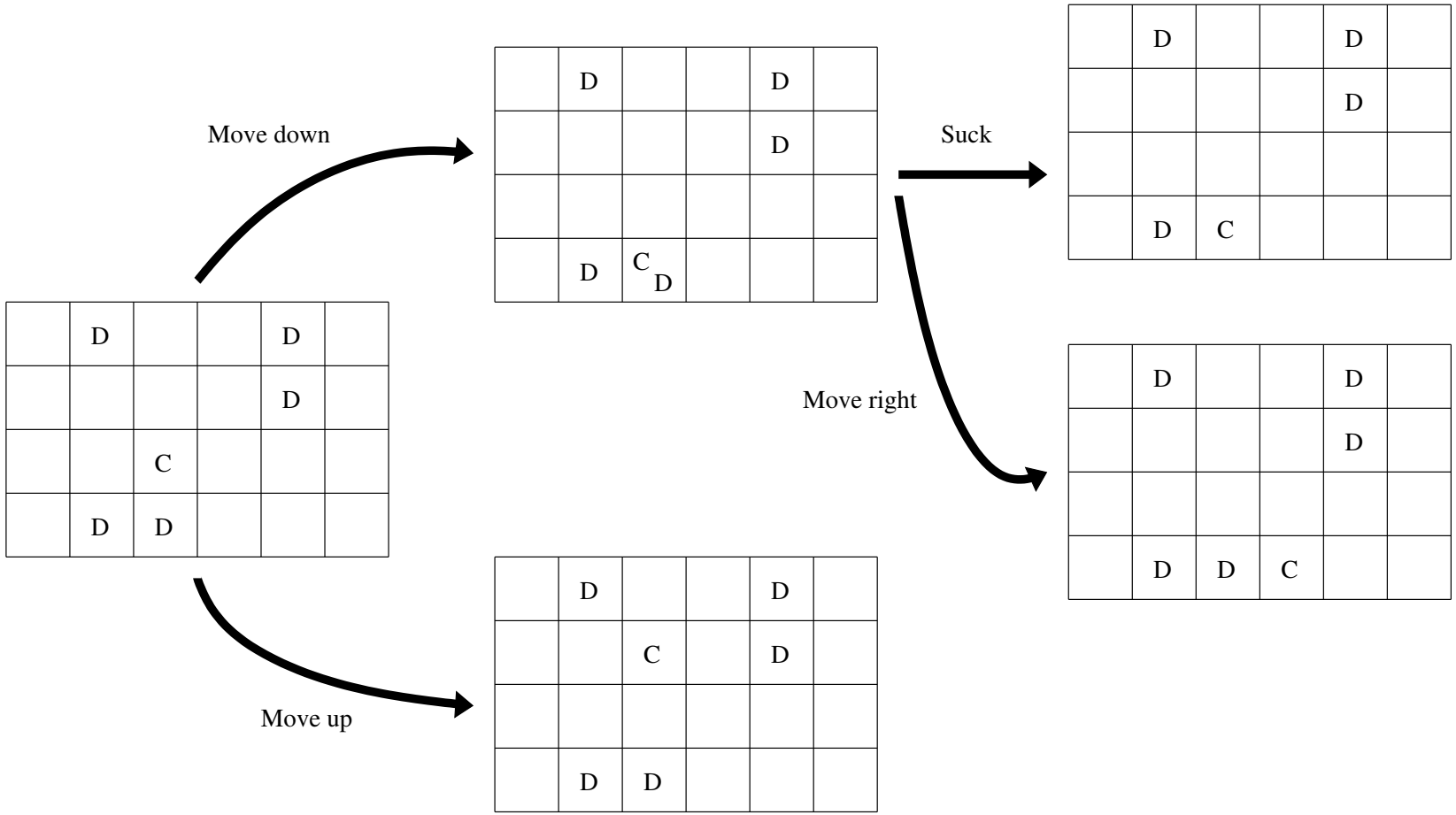
**Initial state:** the cleaner is at some position within the room and there is dirt in various locations.

**Actions:** the cleaner can alter the state of the environment by acting. In this case either by moving or using its sucker. By performing a sequence of actions it can move from state to state.

**Aim:** the cleaner wants to find a sequence of actions that achieves the goal state of having a dirt-free room.

Other applications that can be addressed: route-finding, tour-finding, layout of VLSI systems, navigation systems for robots, sequencing for automatic assembly, searching the internet, design of proteins *etc*.

# Problem solving by basic search



# Problem solving by basic search

So what's ambiguous and subtle here?

1. Can the agent know it's current state in full?
   - It may only be able to sense dirt within a given radius.
   - It may not have a completely accurate position sensor.
   - It may not be able to distinguish between dirt and a stain on the carpet, and so on...
2. Can the agent know the outcome of its actions in full?
   - The sucker may not be completely reliable.
   - The sucker may occasionally deposit a little dirt.
   - The next door neighbour's child may sneak in and move it from one place to another while it thinks it's only moved a short way in one direction, and so on...

# Problem solving by basic search

Depending on the answers to these questions we can identify four basic kinds of problem:

**Single-state problems:** the state is always known precisely, as is the effect of any action. There is therefore a single outcome state.

**Multiple-state problems:** The effect of any action is known, but the current state can not reliably be inferred. Hence we must reason about the set of states that we could be in. A similar situation arises if we know the current state but not necessarily the outcomes of the actions.

Single and multiple state problems can be handled using the search techniques to be discussed next.

**Contingency problems:**

In some situations it is necessary to perform sensing *while* the actions are being carried out in order to guarantee reaching a goal.

(It's good to keep your eyes open while you cross the road!)

This kind of problem requires *planning* and *acting*.

Sometimes it is actively beneficial to act and see what happens, rather than to try to consider all possibilities in advance in order to obtain a perfect plan.

**Exploration problems:**

Sometimes you have *no* knowledge of the effect that your actions have on the environment.

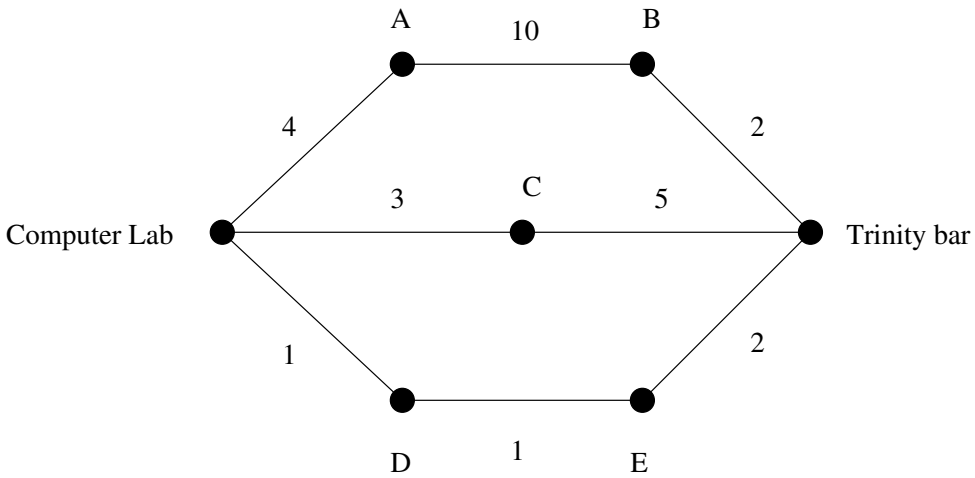
Babies in particular have this experience.

This means you need to experiment to find out what happens when you act.

This kind of problem requires *reinforcement learning* for a solution. We will not cover reinforcement learning in this course.

# Problem solving by basic search

**Question:** How much detail should the state description include?




To use a different example: considering your lecturer as an intelligent agent who wants to get from the Computer Lab to Trinity College before the bar closes.

# Problem solving by basic search

The state of my environment could be said to include:

- the number and temperature of each hair on my head;
- the composition of the roads on all the potential routes;
- the current position of Saturn *etc*.

However for this problem a much simpler state description seems appropriate: “at the computer lab”, “in the computer lab bike stands” and so on.

Similarly for potential actions: although “remove wax from ears” is a perfectly valid action in state “at the computer lab” it’s clearly not very helpful.

**Question:** Are there conflicting goals or goals of varying importance?

Apart from getting to the College bar I might want to stop by the book signing at Waterstones, or drop by the language school to improve my Italian.

However we need to identify one specific goal.

Always in computer science, we need to do some *abstraction* to make a solution feasible—we need to remove all extraneous detail.

Note that in this example, if I have no internal map of Cambridge town centre I am stuck - I am doomed to try random actions. However if I have such a map I can try to *search* for a sequence of actions that achieves my goal.

# Problem solving by basic search

We begin with (arguably) the simplest kind of scenario in which some form of computationally intelligent behaviour can be achieved. Namely, the single-state scenario.

To summarise, we have:

- **an initial state**: what is the agent's situation to start with;
- **a set of actions**: and we know what state will result on performing any available action from any known state;
- **a goal test**: we can tell whether or not the state we're in corresponds to the goal.

Note that the goal may be described by a property rather than an explicit state or set of states, for example "checkmate".

## Problem solving by basic search

In addition, a *path* is a sequence of actions that lead from state to state.

We may also be interested in the *path cost* as some solutions might be better than others. Path cost is generally denoted by $g$.

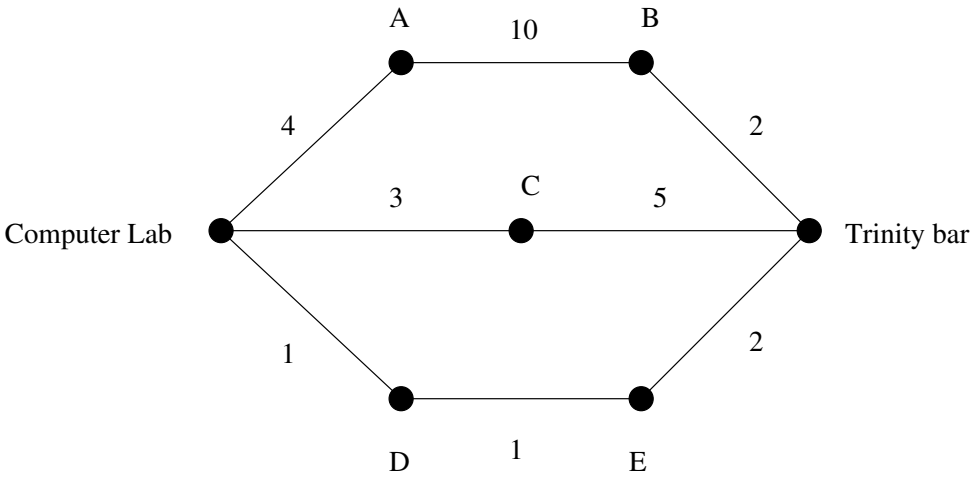
A *solution* is a path beginning with the initial state and ending in a goal state.

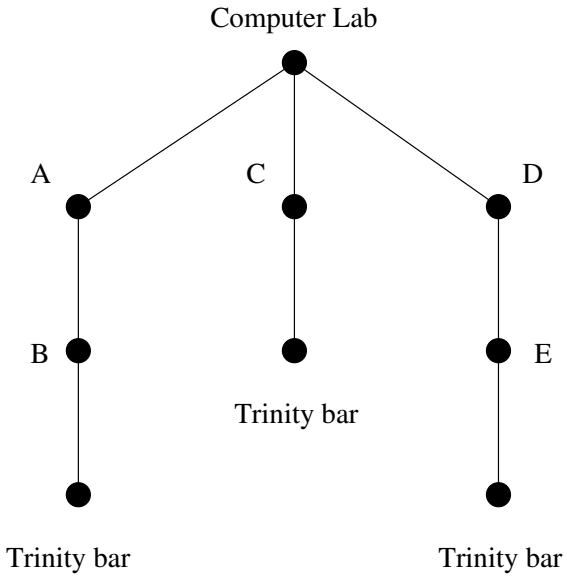
All of the search techniques to be presented can also be applied to multiple-state problems.

- In this case we have an initial *set* of states.
- Each action leads to a further set of states.
- The goal is a set of states *all* of which are valid goals.

# Search trees

The basic method is familiar from your algorithms course.




We form a *search tree* with the initial state as the root node.

# Search trees

Test the root to see if it is a goal.

If not then *expand* it by generating all possible successor states according to the available actions.

If there is only one outcome state then move to it. Otherwise choose one of the outcomes and expand it. The way in which this choice is made defines a *search strategy*.

If a choice turns out to be no good then you can go back and try a different alternative.

The collection of unexpanded states is called the *fringe* or *frontier* and is generally stored as a queue.

# The performance of search techniques

We are interested in:

- whether a solution is found;
- whether the solution found is a good one in terms of path cost;
- the cost of the search in terms of time and memory.

$$\text{the total cost} = \text{path cost} + \text{search cost}$$

If a problem is highly complex it may be worth settling for a sub-optimal solution obtained in a short time.

Other characteristics of the problem may also be relevant. For example I may not want to spend a huge amount of time working out how to get to Trinity.

# Evaluation of search strategies

We are also interested in:

**Completeness:** does the strategy guarantee a solution is found?

**Time complexity**

**Space complexity**

**Optimality:** does the strategy guarantee that the *best* solution is found?

# Search trees

Two types of search:

- *Uninformed* or *blind* search is applicable when we *only* distinguish goal states from non-goal states.

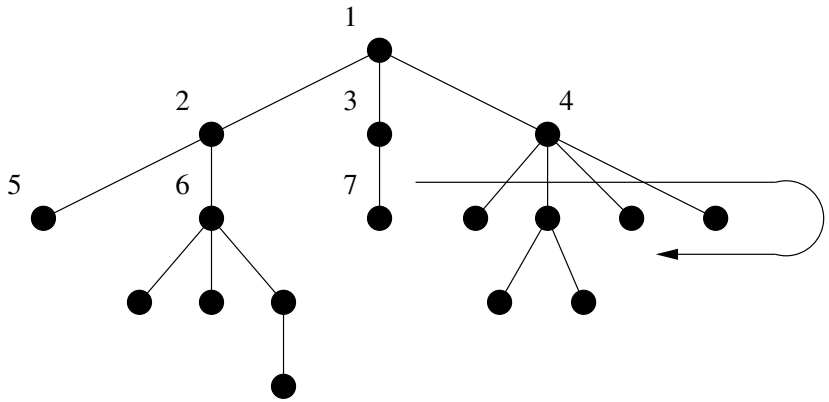
  Methods are distinguished by the order in which nodes in the search tree are expanded. These methods include: breadth-first, uniform cost, depth-first, depth-limited, iterative deepening, bidirectional.

- *Informed* or *heuristic* search is applied if we have some knowledge of the path cost or the number of steps between the current state and a goal.

  These methods include: best first, greedy, A*, iterative deepening A* (IDA*), SMA*.

# Breadth-first search

Breadth-first search:




This is familiar from your algorithms courses.

## Breadth-first search

**Note:**

- the procedure is *complete*: it is guaranteed to find a solution if one exists;
- the procedure is *optimal* under a simple condition: if the path cost is a non-decreasing function of node-depth;
- the procedure has exponential complexity for both memory and time. A branching factor $x$ requires

$$1 + x + x^2 + x^3 + \cdots + x^n$$

nodes if the shortest path has depth $n$.

**In practice**: the memory requirement tends to outweigh the time requirement.

# Uniform-cost search

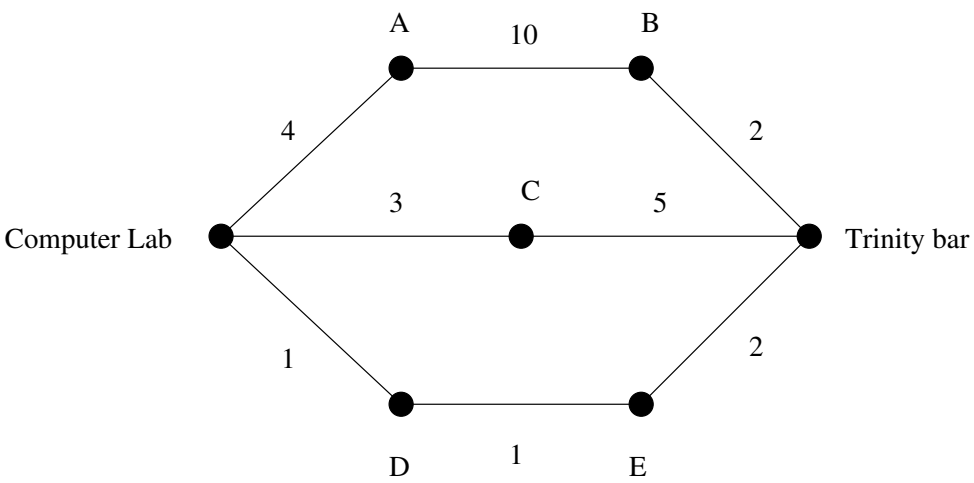
Breadth-first search finds the *shallowest* solution, but this is not necessarily the *best* one.

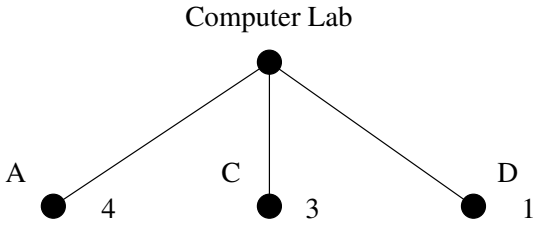
*Uniform-cost search* differs in that it always expands the node with the lowest path-cost $g(n)$ first.

The best solution will always be found if

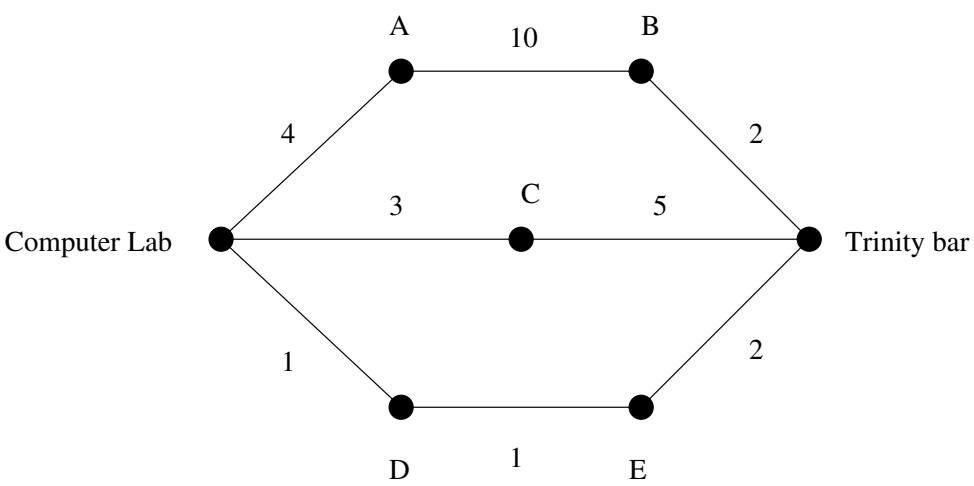
$$\forall \text{node } g(\text{node's successor}) \geq g(\text{node})$$

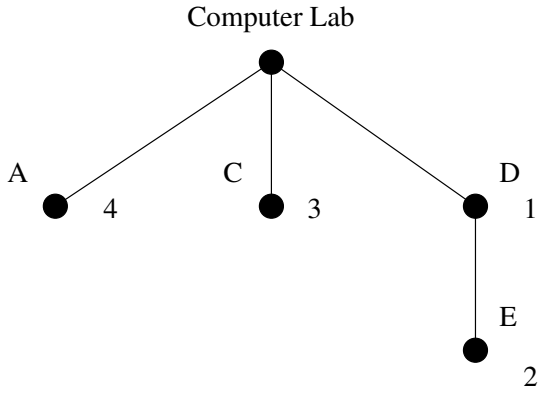
# Uniform-cost search



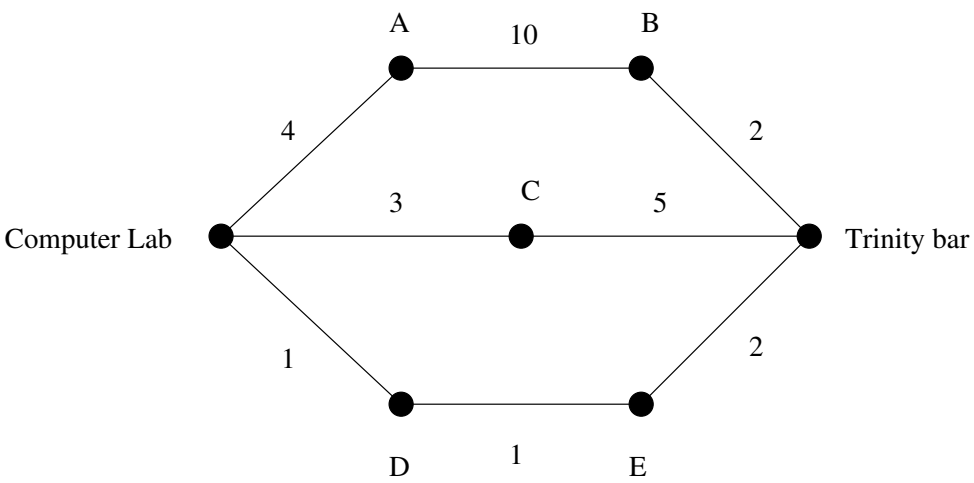



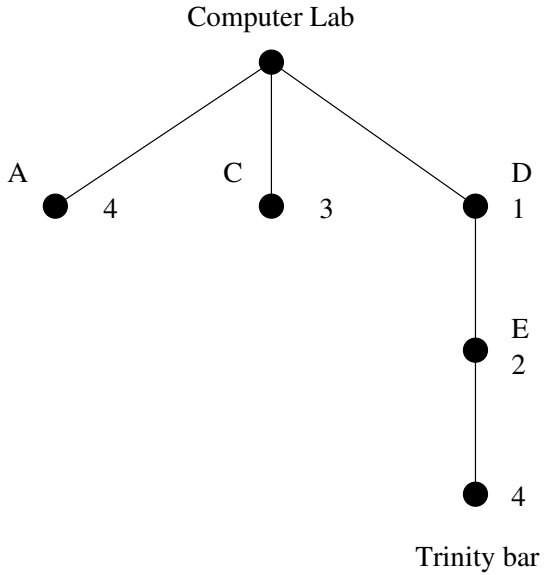
# Uniform-cost search



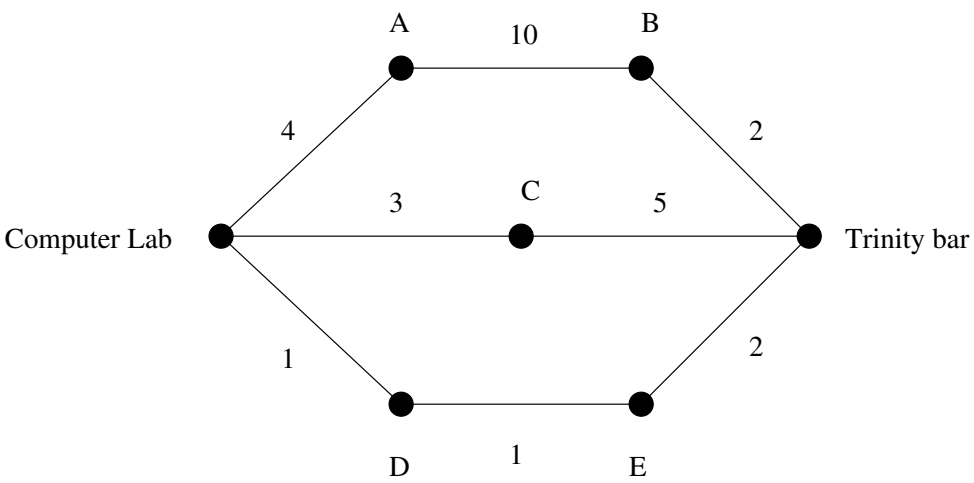



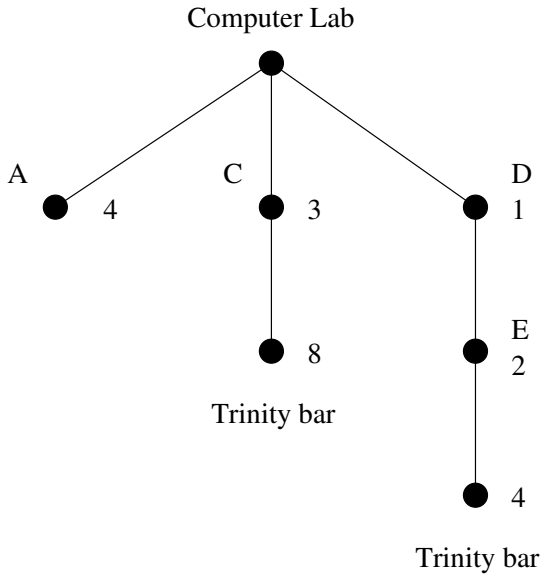
# Uniform-cost search



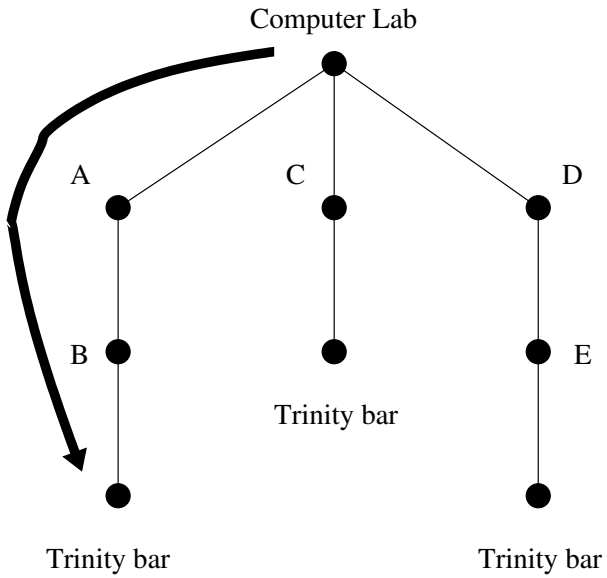



# Uniform-cost search



# Depth-first search




- nodes are expanded at the deepest existing part of the tree;
- for branching factor $x$ and depth $n$ the memory requirement is around $nx$ and the time $O(x^n)$;
- despite the exponential time requirement, if there are *many solutions* this algorithm stands a chance of finding one quickly, compared with breadth-first search.

# Depth-first and depth-limited search

Depth-first search is clearly dangerous if the tree is either very deep or infinite:

- if the tree is very deep we risk finding a suboptimal solution;
- if the tree is infinite we risk an infinite loop.

*Depth-limited search* simply imposes a limit on depth. For example if we're searching for a route on a map with $n$ cities we know that the maximum depth will be $n$. However:

- we still risk finding a suboptimal solution;
- the procedure becomes problematic if we impose a depth limit that is too small.

# Iterative deepening search

Usually we do not know a reasonable depth limit in advance.

*Iterative deepening search* repeatedly runs depth-limited search for increasing depth limits $0, 1, 2, \ldots$

- this essentially combines the advantages of depth-first and breadth-first search;
- the procedure is complete and optimal;
- the memory requirement is similar to that of depth-first search;

Importantly, the fact that you're repeating a search process several times is less significant than it might seem.

# Iterative deepening search

Intuitively, this is because *the vast majority of the nodes in a tree are in the bottom level*:

- in a tree with branching factor $x$ and depth $n$ the number of nodes is
$$f_1(x, n) = 1 + x + x^2 + x^3 + \cdots + x^n$$
- a complete iterative deepening search of this tree generates the final layer once, the penultimate layer twice, and so on down to the root, which is generated $n + 1$ times. The total number of nodes generated is therefore
$$f_2(x, n) = (n + 1) + nx + (n - 1)x^2 + (n - 2)x^3 + \cdots + 2x^{n-1} + x^n$$

## Iterative deepening search

Example:

- for $x = 20$ and $n = 5$ we have

$$f_1(x, n) = 3,368,421$$

$$f_2(x, n) = 3,545,706$$

  which represents a $5$ percent increase with iterative deepening search;

- the overhead gets *smaller* as $x$ increases. However the time complexity is still exponential.

For problems where the search space is large and the solution depth is not known, this is the preferred method.

# Bidirectional search

We can simultaneously search:

*forward* from the *start* state

*backward* from the *goal* state

until the searches meet.

This is potentially a very good idea:

- if the search methods have complexity $O(x^n)$ then...
- ...we are converting this to $O(2x^{n/2}) = O(x^{n/2})$.

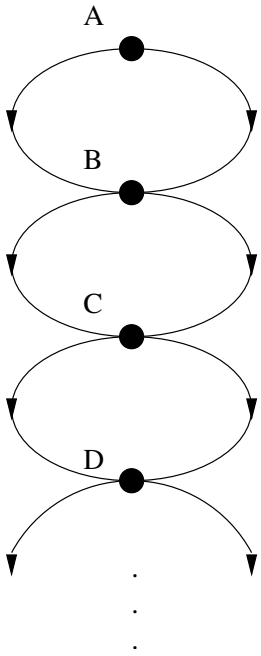
(Here, we are assuming the branching factor is $x$ in both directions.)

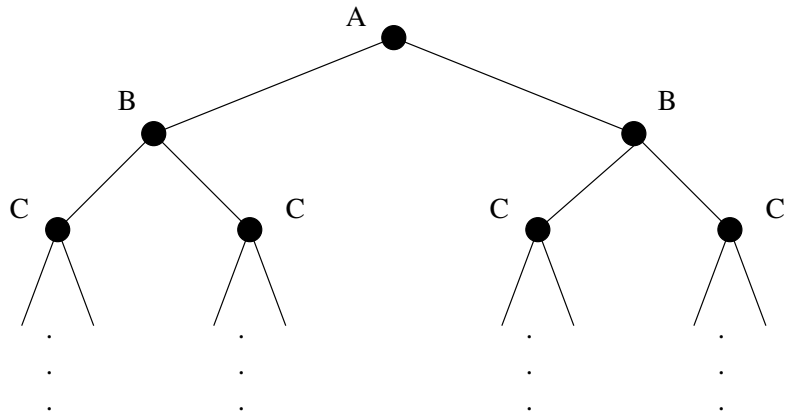
## Bidirectional search

- It is not always possible to generate efficiently *predecessors* as well as successors.
- If we only have the *description* of a goal, not an explicit goal, then generating predecessors can be hard. (For example, consider the concept of *checkmate*.)
- We need a way of checking whether or not a node appears in the other search.
- We need to decide what kind of search to use in each half. For example, would depth-first search be sensible?
- The figure of $O(x^{n/2})$ hides the assumption that we can do constant time checking for intersection of the frontiers. Often this is possible using a hash table.
- To guarantee that the searches meet, we need to store all the nodes of at least one of the searches. Consequently the memory requirement is $O(x^{n/2})$.

# Repeated states

With many problems it is easy to waste time by expanding nodes that have appeared elsewhere in the tree. For example:



# Repeated states

For example, in a problem such as finding a route in a map, where all of the operators are *reversible*, this is inevitable.

There are three basic ways to avoid this, depending on how you trade off effectiveness against overhead.

- never return to the state you came from;
- avoid cycles: never proceed to a state identical to one of your ancestors;
- do not generate *any* state that has previously appeared.