



NUI Galway
OÉ Gaillimh

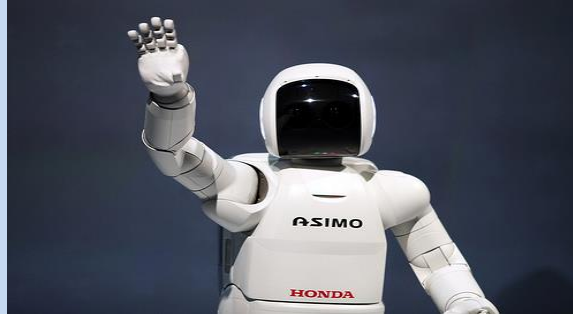


Photo by [Ars Electronica](#) under [CC-BY-NC-ND](#).

AI Search – Uninformed Search

Workshop 2 Section 1

CT621 Artificial Intelligence - MScSED

Welcome to second workshop in Artificial Intelligence (CT621). In this section we examine the concept of *search* in AI focusing specifically on *uninformed search* techniques.

Learning Outcomes

On completing this section and related learning activities, you should be able to:

Formulate

- AI problems

Distinguish

- Between AI search and conventional search

Explain

- The characteristics of uninformed search

Analyse

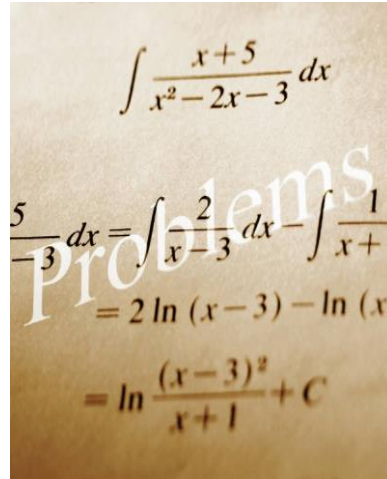
- *Breadth-First Search, Depth-First Search and Iterative Deepening Search*

This workshop will provide an introduction to the area of search within Artificial Intelligence. More specifically, it will examine the concepts of *informed search* and will present search techniques from both categories.

Please review the learning outcomes for this workshop and as you work through the material, check that you are moving towards meeting these.

Solving Problems By Search

- ▶ Artificial intelligence has been successfully used to solve many problems through search
- ▶ Before solving a problem through search we must first formulate the problem



The image shows a handwritten mathematical solution on a piece of paper. The problem is to integrate $\frac{x+5}{x^2-2x-3} dx$. The solution uses partial fraction decomposition, showing the integral as the sum of two simpler integrals: $\int \frac{2}{x-3} dx - \int \frac{1}{x+1} dx$. The final result is $\ln \frac{(x-3)^2}{x+1} + C$.

$$\int \frac{x+5}{x^2-2x-3} dx$$
$$\frac{5}{-3} dx = \int \frac{2}{x-3} dx - \int \frac{1}{x+1} dx$$
$$= 2 \ln(x-3) - \ln(x+1)$$
$$= \ln \frac{(x-3)^2}{x+1} + C$$

[Image Licence CC0](#)

Artificial Intelligence has a long and successful history of solving problems through search. For example, AI search techniques have been applied to solve problems that range from multi-depot time-bounded route planning to robot navigation and even within gaming. This workshop will look at what it means to solve a problem using search and will present various techniques for doing just that. However, before we can solve a problem using an AI search technique we must first have some way of representing or formulating the problem itself. The next few slides describe how that can be achieved.

Formally Defining a Problem in AI

1. Initial State

- The initial state is where the problem starts

2. Possible Actions

- The set of possible actions that can be executed from the state

3. Transition Model

- Returns the state that results from taking an action a in a state s

In AI a problem can be formally defined as consisting of five components:

- 1) **Initial state.** The initial state is where the problem starts. For example, the initial state of a chess problem would be a particular configuration of the chess pieces on the board. The initial state of a routing problem to determine the most efficient route from New York to Washington would be New York.
- 2) **Possible Actions.** A description of all possible actions that can be taken from a given state. For example, in chess there will normally be multiple chess pieces that you can move (multiple possible actions) that will move you from one state to another. Likewise with a routing problem, there are normally multiple different routes (actions) you can take from a given location.
- 3) **Transition Model.** This returns the state that results from taking a particular action. For example, making a move in chess (taking an action) results in a new state (a new configuration of the chess board)

Formally Defining a Problem in AI

4. Goal Test

- Determines if current state is the goal state

5. Path Cost

- A function that assigns a numeric cost to each path

- 4) **Goal Test.** This test determines if a given current state is the goal state. Note there may be multiple goals states and this test will simply check whether the given state is one of them. For example, checkmate is the goal state. Therefore, after I take an action which results in a new state this new state must be checked to determine if we are in the goal state. In the routing problem my goal state would be when I arrive in Washington.
- 5) **Path Cost.** The path cost is a function that assigns a numeric cost to each path where a path is a sequence of states connected by a sequence of actions. For example, in a routing problem time would be an important consideration. Therefore, the cost of a particular path might be its length in kilometres.

Defining the 8-Tile Puzzle

- **State:** A description that specifies the location of each of the numbered tiles on the board
- **Initial State:** Any configuration of the eight numbered tiles
- **Actions:** Movement of the blank space *up, down, left or right*
- **Transition Model:** Returns the configuration after we apply one of the four possible actions
- **Goal Test:** A sequential ordering of the tiles as shown in image on the right
- **Path Cost:** Total number of steps in the path

1	2	3
	4	5
6	7	8

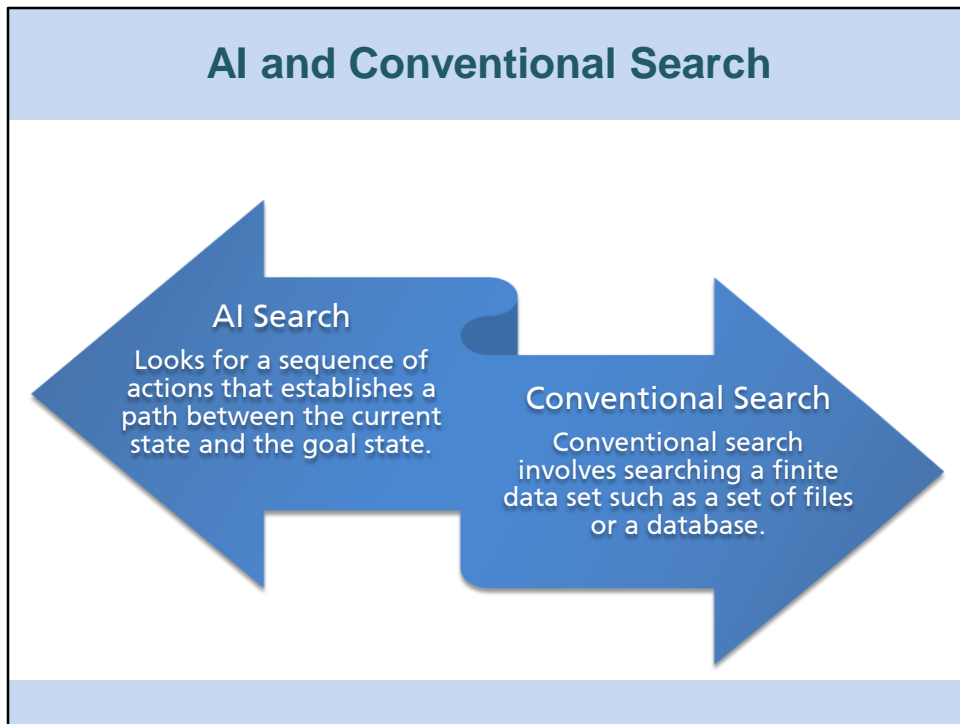
The 8-tile puzzle is composed of a board with 8 numbered tiles and a blank space. Any tile adjacent to the blank space can be moved into the blank space. The objective of the puzzle is to start with the tiles in some random configuration and then move them around until the goal configuration is reached. The goal configuration is generally the arrangement of the tiles in such a way that they are sequentially ordered. The slide shows a possible solution where the tiles are arranged so that they increase in value.

Search Space

*The **search space** is the set of all states reachable from the initial state by any sequence of actions*

*This is also commonly called the **state space***

The initial state, actions and transition model define the *search space* (also referred to as the *state space*) of a problem. The search space forms a directed graph in which each node in the graph is a state and links between the nodes are actions. We will cover this concept in more detail in later slides.



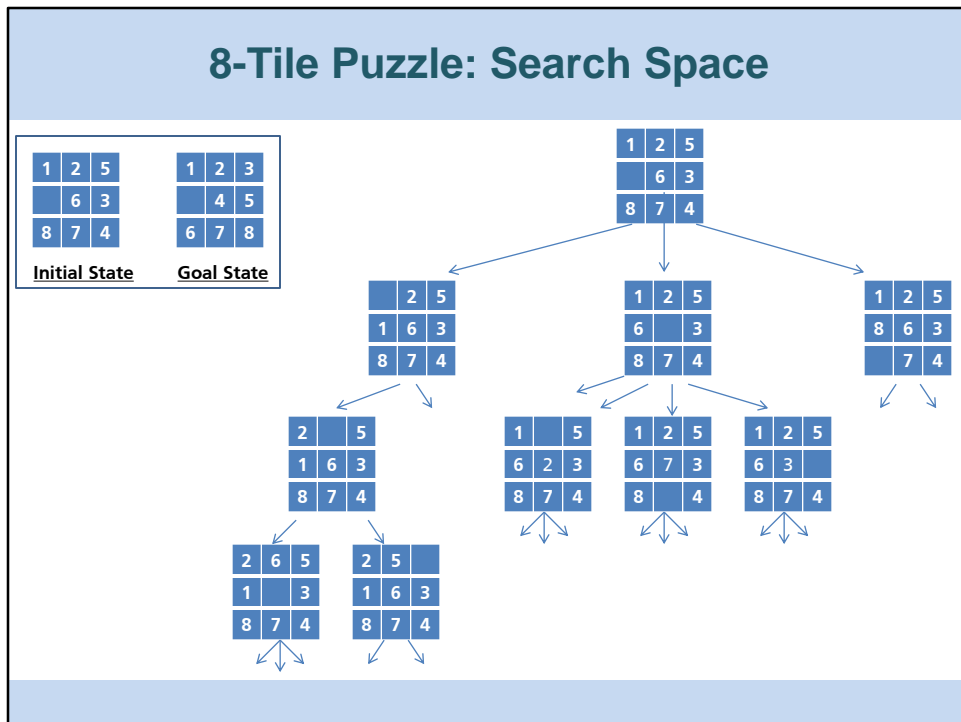
Conventional searching (of files, arrays, databases etc.) is an examination of some data (either structured or unstructured) in an attempt to try and answer a question. For example when you login to an account on a website, a search is made of that site's database, to check that you are a user and that you have entered a valid password. Conventional search involves searching a finite data set, to ascertain the existence of data that satisfies the goal.

AI search can use similar techniques, but with a different objective, and to solve entirely different problems. Rather than searching through data, AI search involves searches for a sequence of actions that can take us from an initial state to a goal state. This can be viewed as a search through the state space of a problem. The state space of a problem consists of the set of all states reachable from the initial state by a particular sequence of actions.

It is important to highlight a key difference between conventional search and AI search. As has already been stated, conventional search involves looking through a set of data. This data is finite and organised (presumably), so it is possible to examine all of the data. AI search on the other hand is searching for a sequence of actions that take us from the initial state through potentially multiple different states until we arrive at a goal state. These states only being generated as required. In other words the search space is virtual, as only those states that are considered are created. One consequence of this is that AI search can work in effectively infinite search spaces, which would be meaningless for conventional search (as an infinite quantity data is impossible).

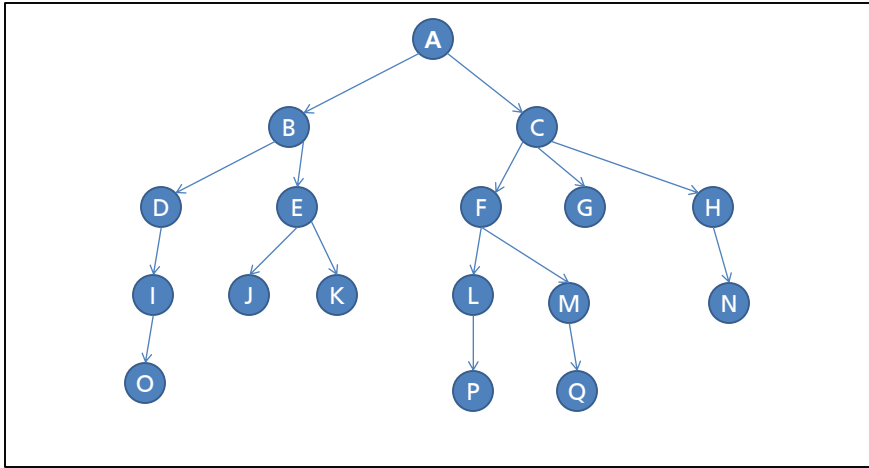
8-Tile Puzzle: Searching for a solution

- ▶ The following slide shows part of the state space for the 8-tile puzzle
- ▶ Each state represents the result of a **potential action** (not necessarily an actual action taken)



The tree depicted in the slide shows some of the possible actions that can be taken to move from one state to another as the search attempts to find a path from the initial state to the goal state. Notice that an action that takes us from one state to an adjacent states requires the movement of the blank space up, down, left or right by one position .

AI Search



We've now seen the representational concepts of a state and actions that link different states. The rest of this workshop will focus on the various techniques that can be used to search the space.

To illustrate these techniques, the simple search space shown in this slide will be used.

For simplicity, the states are each labeled with a letter. The initial state is labeled **A** and the goal state is labeled **Q**.

For each of the techniques, the aim will be to find a path from **A** to **Q**.

Uninformed Search Strategy

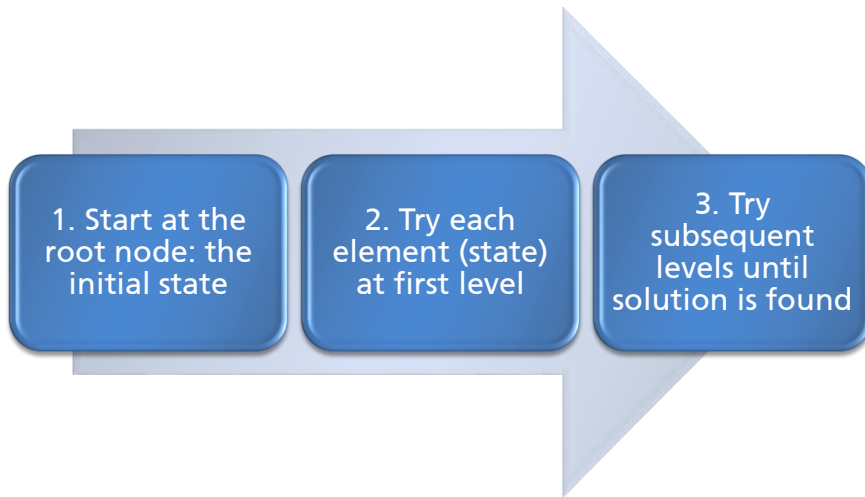
- ▶ Uninformed search (blind search) strategies adopt a rigid search pattern
- ▶ These strategies have no additional information about states beyond what is provided in the problem definition
- ▶ They do not bias their search pattern because one state is more promising than another

Uninformed search strategies, also known as “blind search,” use no information about the possible direction of the goal state. They do not distinguish one state as being superior in any way to another (unless it is the goal state). Therefore, the quality of one state over another has no influence over the search pattern employed by an uninformed search strategy.

Uninformed search strategies include *Breadth-First Search*, *Depth-First Search*, *Depth-Limited Search*, *Uniform-Cost Search*, *Depth-First Iterative Deepening Search* and *Bidirectional Search*.

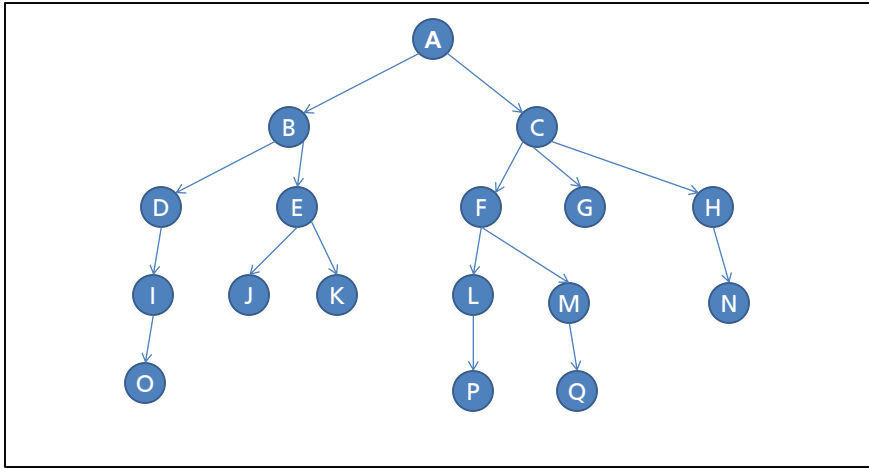
In this workshop we will look at Breadth-first search, Depth-First Search and Depth-First Search with Iterative Deepening.

Breadth-First Search



The first search we will examine is one of the simplest, Breadth-First Search . As its name implies, Breadth-First Search works across the search space, one level at a time. Every state at a level **MUST** be examined before any states at deeper levels can be considered.

Breadth-First Search Example



In the example, states B and C must be examined before any states from D onwards can be considered. So before state **O** can be considered, ALL of the states labeled **A...N** must have been examined.

A consequence of this approach is that Breadth-First Search is guaranteed to find the shortest path from the initial state to the goal state, as deeper levels cannot be examined until the lower levels have been exhausted.

A simple extension to the Breadth-First Search is to record all of the paths taken, this can then be used to recreate the optimal path, once it has been found. Only once a level has been exhausted can lower levels be considered.

Breadth-First Search: Algorithm

```
Procedure BreadthFirstSearch (Tree tree, Node root, Goal
goal, List alreadyVisited)

    Node current
    Queue toVisit

    add root to toVisit

    while toVisit is not empty
        current <- first node on toVisit
        remove first node from toVisit
        if current = goal
            add current to alreadyVisited
            return true
        end if
        for each child node C of current
            add C to toVisit
        end for
        add current to alreadyVisited
    end while
```

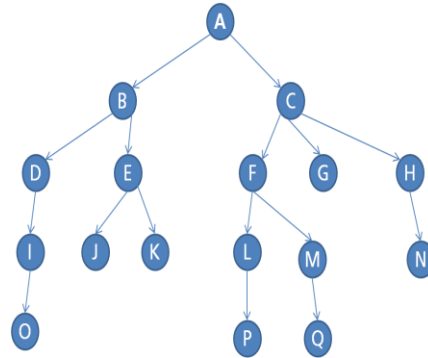
[Image Licence CC0](#)

The Breadth-First Search algorithm essentially keeps a record of states that it knows about, but hasn't visited yet. When it examines a state, it first checks if it is the goal. If it is the search stops, if it isn't it finds all of the children of the current state (all of those states that can be reached by a single transition). These new states are then added to the list of states still to be visited **toVisit**.

As **toVisit** is a queue, all of these new states are added to the end of the queue. The use of a queue allows the state that has been known about the longest to be considered next. This simple use of the queue implements the Breadth-First Search for us.

Breadth-First Search: Search Trace

toVisit	alreadyVisited
[A]	[]
[B, C]	[A]
[C,D,E]	[A, B]
[D, E, F, G, H]	[A, B, C]
[E, F, G, H, I]	[A, B, C, D]
[F, G, H, I, J, K]	[A, B, C, D, E]
[G, H, I, J, K, L, M]	[A, B, C, D, E, F]
[H, I, J, K, L, M]	[A, B, C, D, E, F, G]
[I, J, K, L, M, N]	[A, B, C, D, E, F, G, H]
[J, K, L, M, N, O] ...	[A, B, C, D, E, F, G, H, I] ...



This slide shows the contents of **toVisit** and **alreadyVisited** as the search progresses.

In the first line, no states have been examined yet, **alreadyVisited** is empty, and the only known state is **A** (the initial state). This is not the goal state, so its children are generated and added to the queue. This continues until the goal is reached.

Notice the order of the elements in **alreadyVisited**. By the end of the search, it will contain every element examined (except the goal). The order of the states is also important, this ordering tells us that they were visited in the order A,B,C,D,....,P which is the same as reading across each row in the tree, before moving onto the next.

Notice also, that the number of elements on **toVisit**, gets quite large (up to 7), which is a large proportion of the states in the tree.

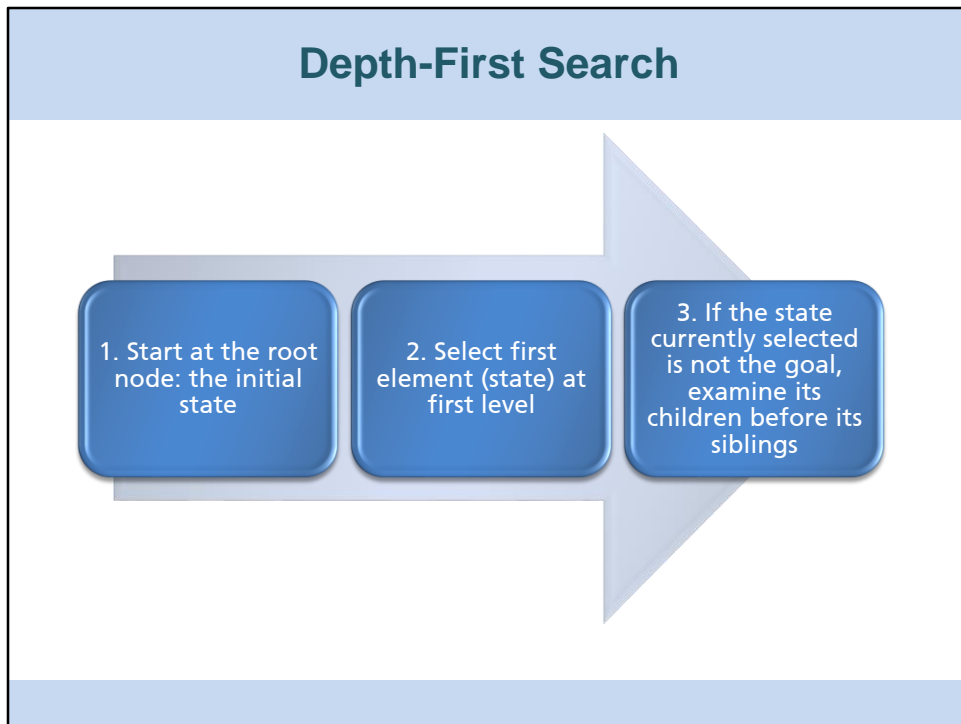
Breadth-First Search: Efficiency

- ▶ Breadth-First Search works well if it is known that a short path exists
- ▶ If trees are deep, with a large branching factor, toVisit list will grow very fast
 - ▶ Exponentially
 - ▶ B^n (B = branching factor, n = depth)

Breadth-First Search must visit every state at the current level in the tree it is examining, and as the levels get deeper, the number of states at that level grows exponentially.

Even for a simple space with a branching factor (B) of 2 at depth 10 there will be 2^{10} states on **toVisit** which is 1024, for larger branching factors and depths it is far worse.

The *branching factor* is the average number of children that each state has.



The next uninformed search technique is Depth-First Search.

Rather than searching an entire level, the search picks a state, and examines that state and all of its children before examining the other states at the current level.

As can be seen on the next slide, the only difference between the Breadth-First and Depth-First Searches is the data structure used to represent **toVisit**. In Breadth-First Search, it was a **queue**, so the oldest nodes were returned first. In Depth-First Search, it is a **stack**, so the newest elements are returned first.

Depth-First Search Algorithm

```
Procedure DepthFirstSearch (Tree tree, Node root, Goal
goal, List alreadyVisited)

    Node current
    Stack toVisit

    add root to toVisit

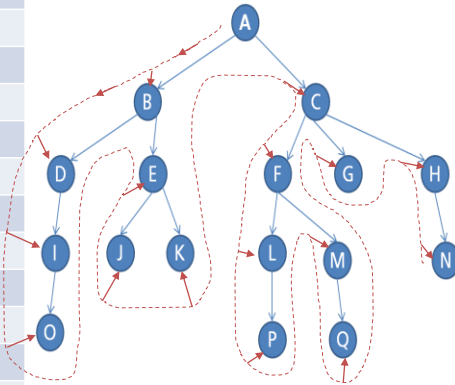
    while toVisit is not empty
        current <- first node on toVisit
        remove first node from toVisit
        if current = goal
            add current to alreadyVisited
            return true
        end if
        for each child node C of current
            add C to toVisit
        end for
        add current to alreadyVisited
    end while
```

[Image Licence CC0](#)

This slide shows the algorithm for Depth-First Search. The algorithm is almost the same as that for Breadth-First Search, except that **toVisit** is represented as a stack, rather than a queue.

Depth-First Search Trace

toVisit	alreadyVisited	
[A]	[]	
[B, C]	[A]	
[D, E, C]	[A, B]	
[I, E, C]	[A, B, D]	
[O, E, C]	[A, B, D, I]	
[E, C]	[A, B, D, I, O]	
[J, K, C]	[A, B, D, I, O, E]	
[K, C]	[A, B, D, I, O, E, J]	
[C]	[A, B, D, I, O, E, J, K]	
[F, G, H]	[A, B, D, I, O, E, J, K, C]	
[L, M, G, H]	[A, B, D, I, O, E, J, K, C, F]	
[P, M, G, H]	[A, B, D, I, O, E, J, K, C, F, L]	
[M, G, H]	[A, B, D, I, O, E, J, K, C, F, L, P]	
[Q, G, H]	[A, B, D, I, O, E, J, K, C, F, L, P, M]	



This slide shows the behaviour of the Depth-First Search. **toVisit** and **alreadyVisited** represent the same information as before.

Three things to notice about this figure:

- 1 The number of elements in **toVisit** never exceeds 4 (it rose to 6 for Breadth-First Search).
- 2 The order that the states are visited in is different from Breadth-First (as can be seen from **alreadyVisited**).
- 3 Not all of the states have been examined: **G** and **H** are on **toVisit** and **N** hasn't even been found (as **H** hasn't been examined).

The red path overlay shows the order in which the nodes are visited.

Depth-First Search: Efficiency

- Depth-First tries 'deep' branches before shallow ones
- Path may not be found by optimal route
- However, deep trees with a large branching factor do not pose a problem as the open list only grows linearly ($B * n$ (B average branches, n depth))
- Depth-First Search may get stuck if the search space is infinitely deep

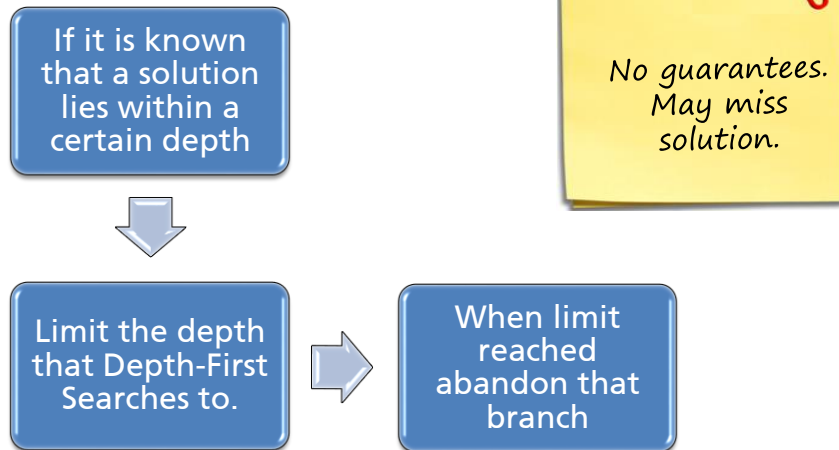
As Depth-First Search effectively burrows into the search space, its behaviour is different from Breadth-First Search.

Whilst Breadth-First Search is guaranteed to find the optimal path, Depth-First Search may find a sub-optimal path.

The advantage that Depth-First Search has over Breadth-First Search is its complexity. For Breadth-First Search, the size of **toVisit** was B^n , whilst for Depth-First Search it is only $B * n$. for a branching factor of 2 and a depth of 10, this gives the size of **toVisit** as 20, compared to 1024 for the Breadth-First Search.

Whilst both techniques have their advantages, what we want is a technique that is guaranteed to find the optimal path and is efficient.

Depth-Bounded Depth-First Search



[Image Licence CC0](#)

The first attempt at overcoming the limitations of both Depth-First and Breadth-First Search, whilst having all of their advantages, is *Depth-Bounded Depth-First Search*.

This search, as its name implies is a version of Depth-First Search. Rather than searching to deeper and deeper depths, a limit is placed on the maximum depth that can be searched to. If the search reaches the threshold depth without finding the goal, that branch is abandoned.

This approach overcomes one of the problems of Depth-First Search in that it can't get stuck in infinitely deep search spaces. However, it isn't guaranteed to find the shortest path. Even worse, if the shortest path is longer than the applied threshold, the path cannot be found at all.

Depth-First Search with Iterative Deepening

```
Set max_depth = 1
```

```
Repeat:
```

```
    do depth-bounded Depth-First Search to level  
    max_depth  
    max_depth = max_depth + 1
```

```
Until solution found
```

[Image Licence CC0](#)

The algorithm for **Depth-First Search with iterative deepening** shown here overcomes all of the problems of Depth-First Search and Breadth-First Search and at the same time has all of the advantages.

The method works by performing depth-bounded Depth-First Search to a depth of 1; if the goal is found then the algorithm stops. If the goal is not found, the depth threshold is increased by one and the bounded Depth-First Search is repeated. This carries on until the goal is found.

Depth-First Search with Iterative Deepening

- Guaranteed to find shortest path
- Despite apparent large repetition same order of complexity as depth and Breadth-First $O(B^n)$
- Most of search is in low depths
- Space usage is $B \cdot n$ (B = branching factor, n = depth)
- Result is an efficient algorithm that is guaranteed to find the shortest path to the goal

Image Licence CC0

This method is guaranteed to find the shortest path, as level $n+1$ is not examined until ALL of level n has been examined (in the previous iteration), which gives it an advantage over Depth-First Search.

It may appear to be more computationally complex than both depth and Breadth-First Search. However as most of the repeated searching is at a low level, there is not much difference between them. In fact the order of complexity is exactly the same $O(B^n)$, as all of the states may need to be considered. (Note: Big-O notation is covered in CT609 - Fundamentals of Programming.)

In terms of space usage, it only ever performs a Depth-First Search so its space usage is $B \cdot n$, this gives it an advantage over Breadth-First Search.

In summary, **Depth-First Search with iterative deepening** is both efficient and is guaranteed to find the shortest path and so of all the algorithms we have met so far it is the best.



Self-Review Question: Uninformed Search

Quiz - 1 Question

Last modified: Friday, October 12, 2018 at 11:01:35 AM

Properties

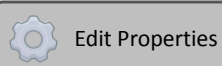
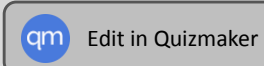
On passing, 'Finish' button: [Goes to next slide](#)

On failing, 'Finish' button: [Goes to next slide](#)

Allow user to leave quiz: [At any time](#)

User may view slides after quiz: [Any time](#)

Show quiz in menu as: [Multiple Items](#)



Summary

AI Search

Examined the core components of representing a problem in AI (Initial State, Actions, Transition Model, Goal State and Path Cost)

Compared and contrasted AI search with conventional search

Uninformed Search

Presented the concept of uninformed search

Analysed the operation and efficiency of Breadth-First Search, Depth-First Search and Depth-First Iterative Deepening Search

References

ArsElectronica (2010) ASIMO [photo], ([CC BY-NC-ND 2.0](#)), available:
<https://www.flickr.com/photos/36085842@N06/4945217670/> [accessed 18 Apr 2019].

All CC0 licensed images were accessed from Articulate 360's Content Library