# Artificial Intelligence

# DFS and BFS

Zahoor Tanoli (PhD)
CUI Attock Campus

## You will learn

- What is search and why it is needed
- What is uninformed searches
- Understanding and implementing DFS and BFS
- Time and space complexity
- Informed search introduction

## Simple Search Algorithm

The process of deciding

- what actions and states to consider

E.g., driving Attock → Lahore

- in-between states and actions defined
- States: Some places in Attock & Lahore
- Actions: Turn left, Turn right, go straight, accelerate & brake, etc.

## Search Algorithm

Because there are many ways to achieve the same goal

- Those ways are together expressed as a tree
- Multiple options of unknown value at a point,
  - the agent can examine different possible sequences of actions, and choose the best
- This process of looking for the best sequence is called ***search***
- The best sequence is then a list of actions, called ***solution***

## Search Algorithm

- Defined as
  - taking a *problem*
  - and returns a *solution*
- Once a solution is found
  - the agent follows the solution
  - and carries out the list of actions – execution phase
- Design of an agent
  - “Formulate, search, execute”

## Simple Search Algorithm

A search node is a path from some state X to the start state, e.g., (XBAS)

The state of a search node is the most recent state of the path, e.g. X.

Let Q be a list of search nodes, e.g. ((XBAS) (CBAS)...)

Let B be the start state

1. **Initialize Q with search node (S) as only entry; set Visited = (S)**
2. **If Q is empty, fail. Else, pick some search node N from Q**
3. **If state (N) is a goal, return N (we've reached the goal)**
4. **(otherwise) Remove N from Q**
5. **Find all the descendants of state(N) not in Visited and create all the one-step extensions of N to each descendant.**
6. **Add the extended paths to Q; add children of state(N) to Visited**
7. **Go to step 2.**

### Critical Decisions:

**Step 2: Pick N from Q**

**Step 6: adding extension of N to Q**

## Implementing the Search Strategies

**Depth-First**

- **Pick first element of Q**
- **Add path extensions to front of Q**

**Breadth-first**

- **Pick first element of Q**
- **Add path extensions to end of Q**

## Testing for the goal

- This algorithm stops (in step 3) when state(N)=G or, in general, when state(N) satisfies the goal test.
- We could have performed this test in step 6 as each extended path is added to Q. This would catch termination earlier and be perfectly correct.
- However, performing the test in step 6 will be incorrect for the optimal searches. I have chosen to leave the test in step 3 to maintain uniformity with these future searches.

## Definition

### Visited *(not expanded)*

**a state M is first visited when a path to M first gets added to Q. In general, a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly "visited" them to place them on Q, but we have not yet examined them carefully.**

### Expanded

**a state M is expanded when it is the state of the search node that is pulled off of Q. At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M (instead of M itself) as being expanded. However, once a node is expanded we are done with it. We will not need to expand it again. In fact, we discard it form Q.**

## Visited States

**Keeping track of visited states generally improves time efficiency when searching graphs, without affecting correctness.**

*Space may still be a problem to keep track of all visited states*

**If all we want to do is find a path from the start to the goal, there is no advantage to adding a search node whose state is already the state of another search node**

**This would be the ideal way of avoiding loops in search graphs/trees**

**Any state reachable from the node the second time would have been reachable from that node the first time**

**Note that, when using Visited, each state will only ever have at most one path to it (search node) in Q**

## Implementation Issues

- **Although we will be speaking about Visited list, this is never the preferred implementation – needs continuous checking if any node is present in the list**
- **If the graph states are known ahead of time, as an explicit set then space is allocated in the state itself to keep a mark; which makes both adding to Visited and checking if a state is Visited a constant time operation – like if we have a data structure we can just add a flag bit**
- **Alternatively, as is more common in AI, if the states are generated on the fly, then a hash table may be used for efficient detection of previously visited states – it still require some space and constant access time**
- **Note that, in any case, the incremental space cost of the Visited list will be proportional to the number of states – which can be very high in some problems**

## Terminology

**Heuristic –** **The word generally refers to a "rule of thumb" something that may be helpful in some cases but not always. Generally held to be in contrast to "guaranteed" or "optimal"**

**Heuristic Function –** **In search terms, a function that computes a value for a state (but does not depend on any path to that state) that may be helpful in guiding the search.**

**Estimated Distance to Goal –** **this type of heuristic function depends on the state and the goal. The classic example is straight-line distance used as an estimate for actual distance in a road network. This type of information can help increase the efficiency of the search**

## Implementing the Search Strategies

### Depth first

Pick first element of Q
Add path extensions to front of Q

### Breadth first

Pick first element of Q
Add path extensions to end of Q

### Best first (Informed) or greedy *uses heuristics*

*Not guaranteed of finding the best or optimal path ☹*

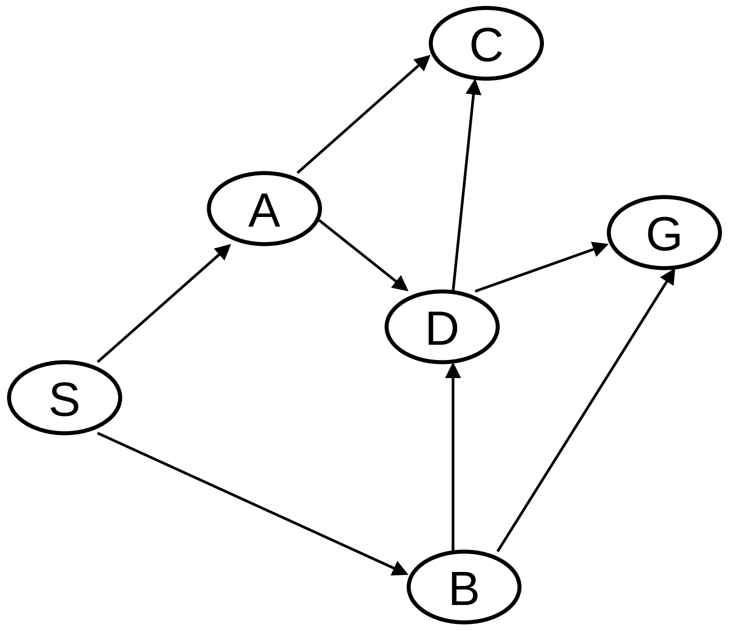
Pick "best" (measured by heuristic value of state) element of Q
Add path extensions anywhere in Q (It may be more efficient to keep the Q ordered in some way so as to make it easier to find the "best" element

**In worst case it would examine all the same paths that DFS and BFS would examine – obviously, order will not be the same and so do the path**

# Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | Q | Visited |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |



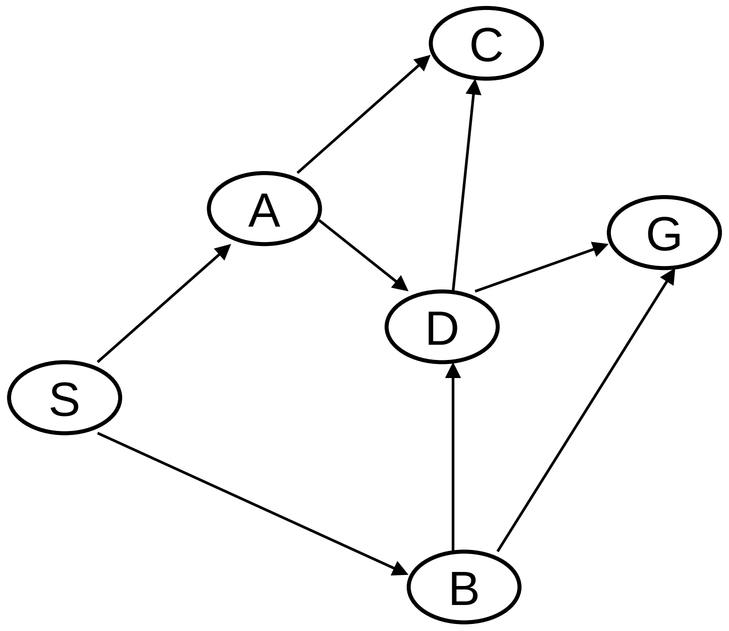
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

# Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |



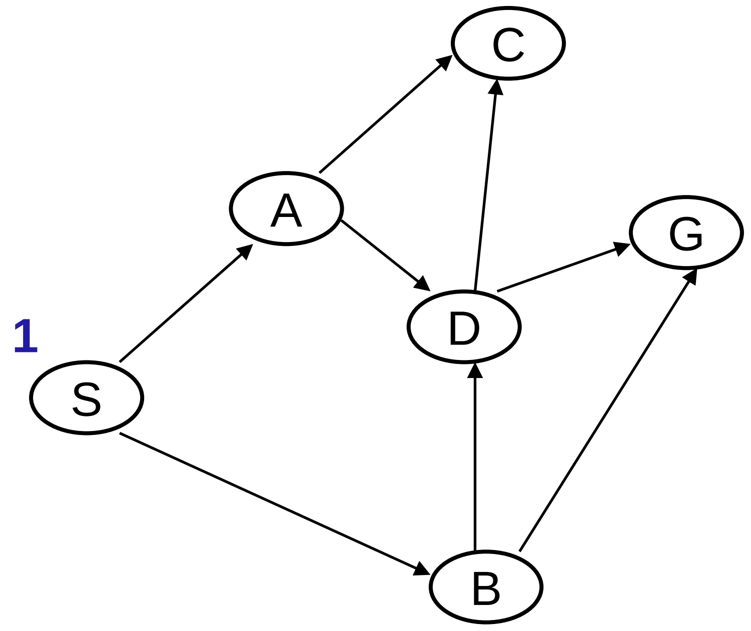
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | (S) | S |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |



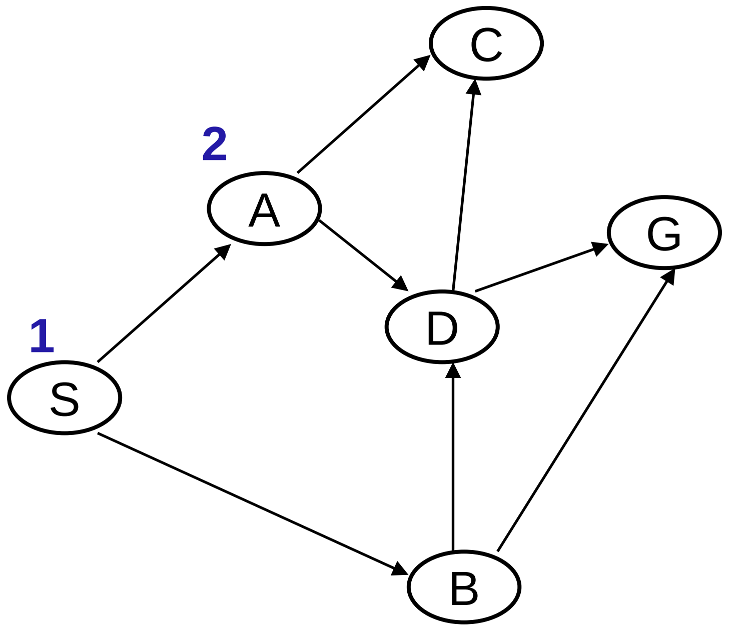
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | | |
| 4 | | |
| 5 | | |



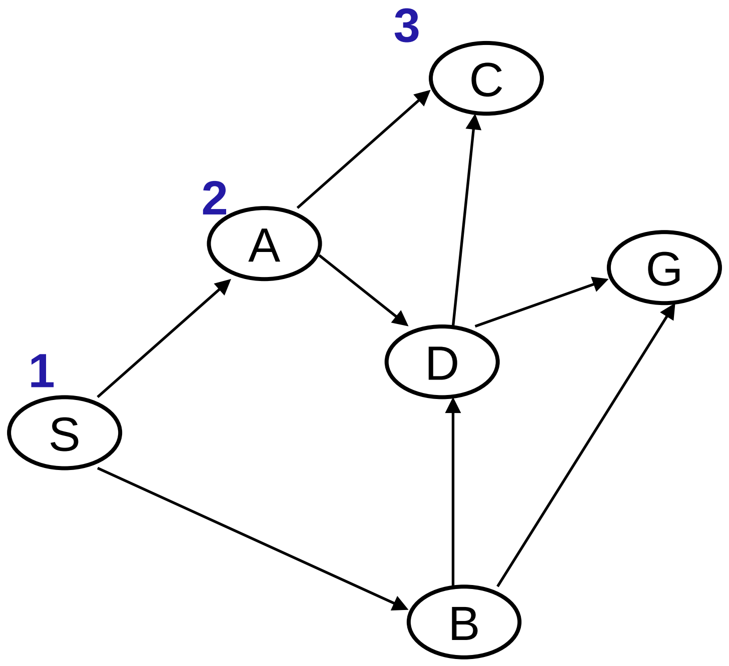
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | **(C A S) (D A S) (B S)** | **C, D, B, A, S** |
| 4 | | |
| 5 | | |



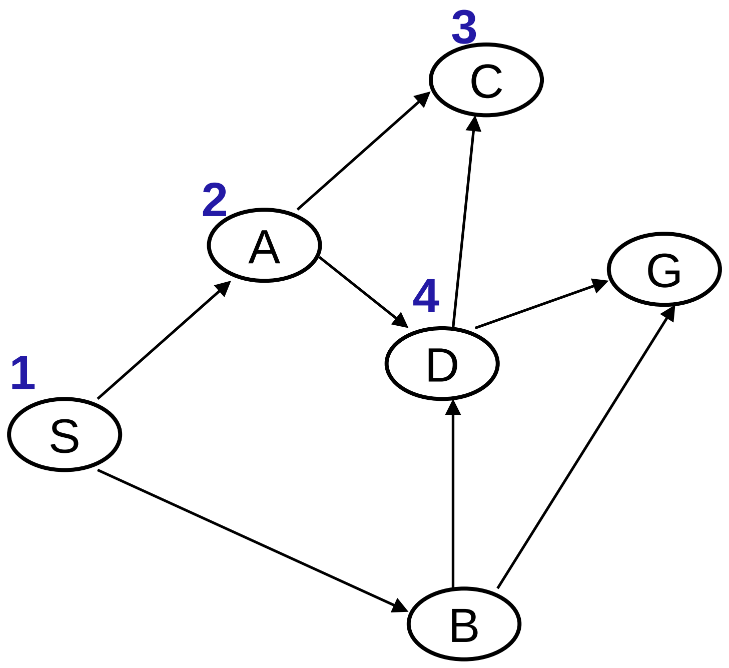
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

# Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | **(C A S) (D A S) (B S)** | **C, D, B, A, S** |
| 4 | **(D A S) (B S)** | **C, D, B, A, S** |
| 5 | | |



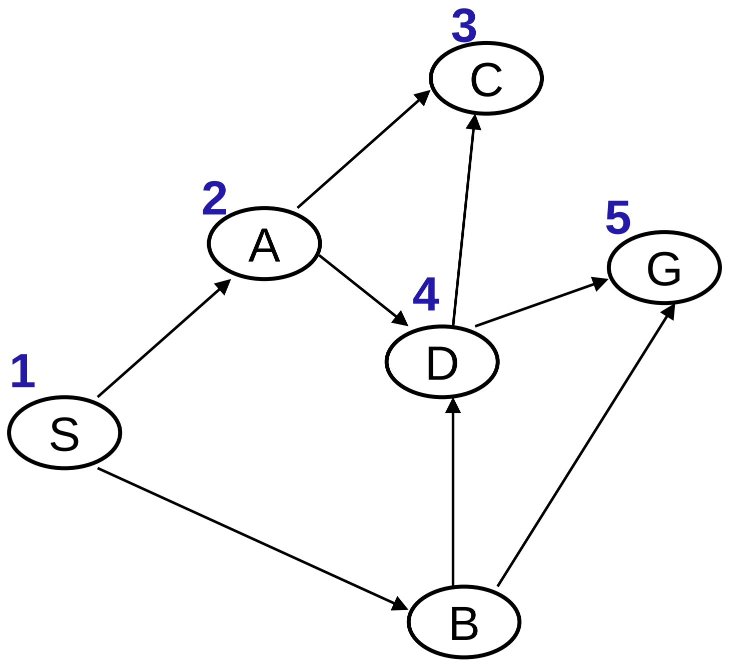
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (C A S) (D A S) (B S) | C, D, B, A, S |
| 4 | (D A S) (B S) | C, D, B, A, S |
| 5 | (G D A S) ( B S) | G, C, D, B, A, S |



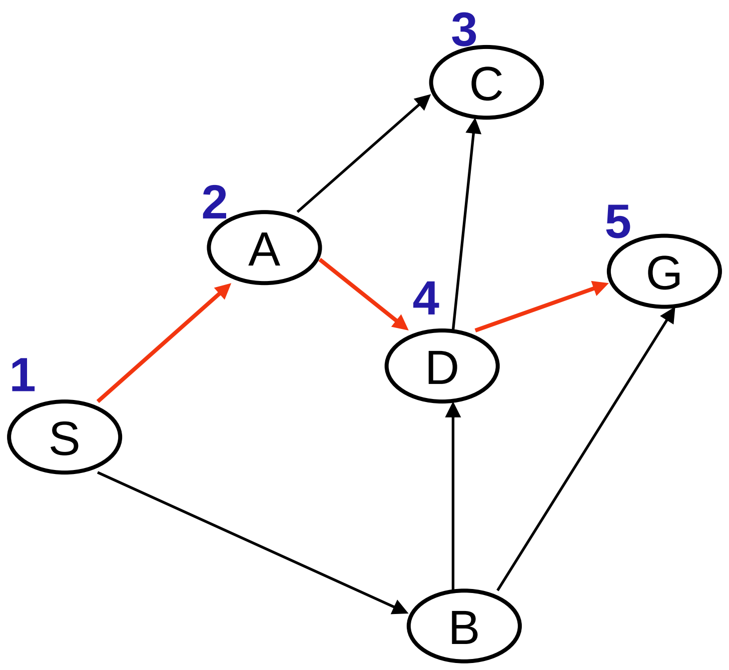
**Added paths in blue**

**Path is shown in reversed order, the node’s state is the first entry**

# Depth First Search

**Pick First element of Q; Add path extensions to front of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | **(C A S) (D A S) (B S)** | **C, D, B, A, S** |
| 4 | **(D A S) (B S)** | **C, D, B, A, S** |
| 5 | **(G D A S) ( B S)** | **G, C, D, B, A, S** |

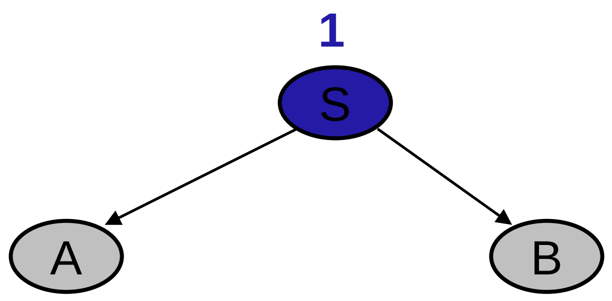


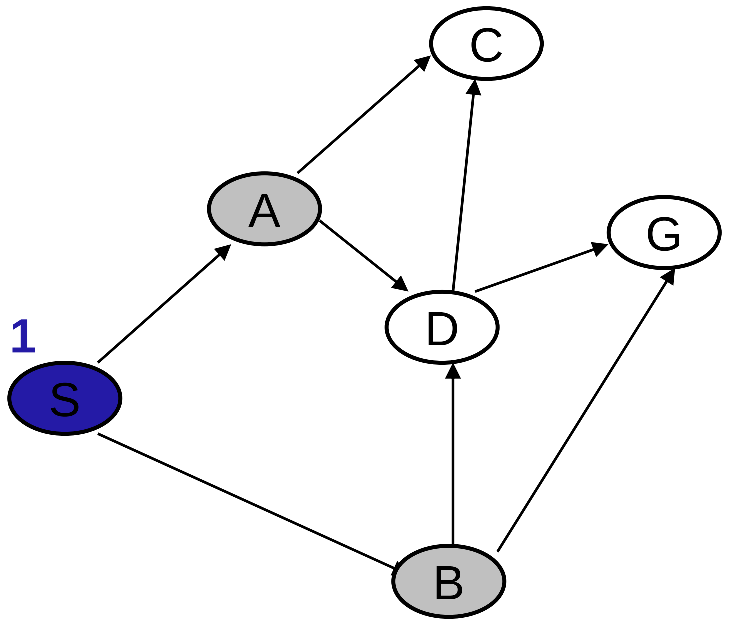
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Depth First Search

Also we have look at this as,





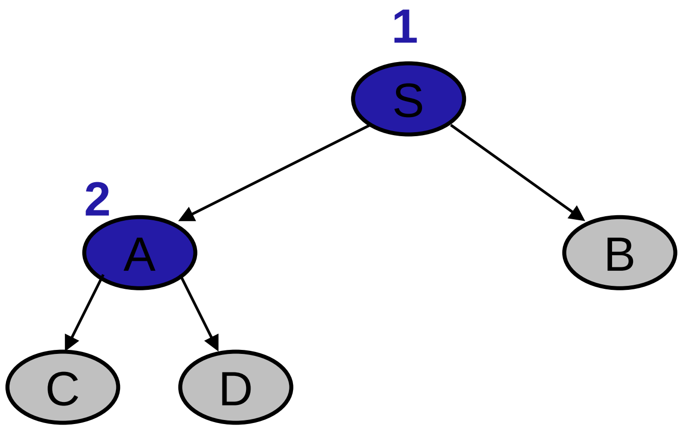
**Numbers indicate order pulled off of Q (expanded)**

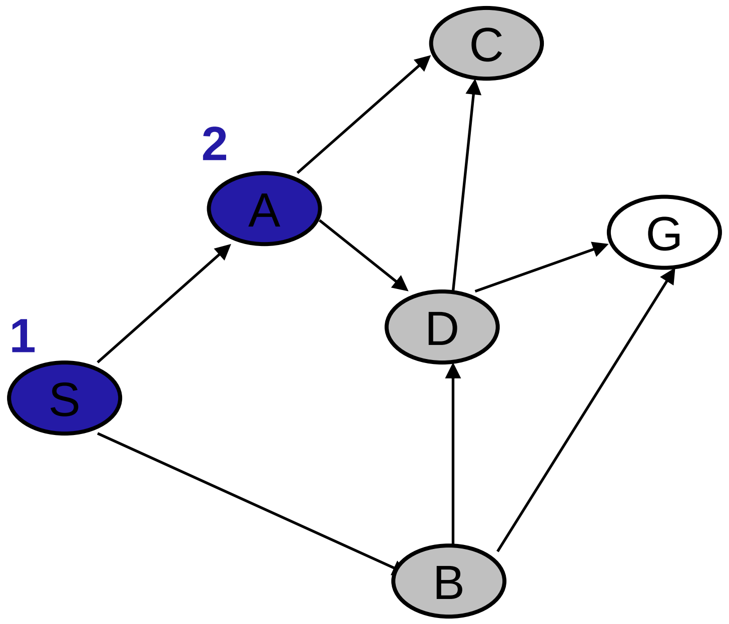
**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

## Depth First Search

Also we have look at this as,





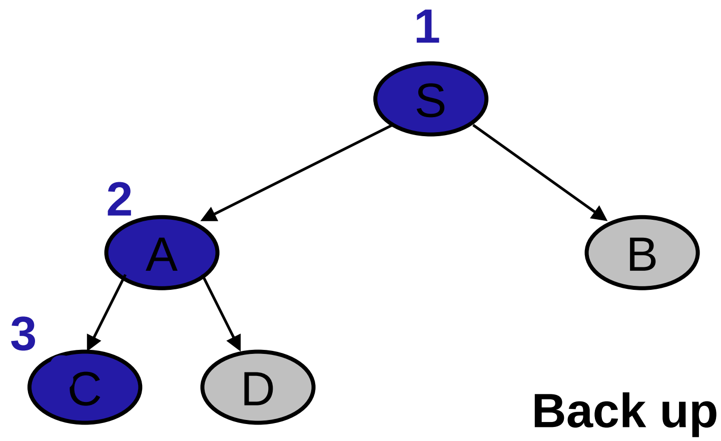
**Numbers indicate order pulled off of Q (expanded)**

**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

## Depth First Search

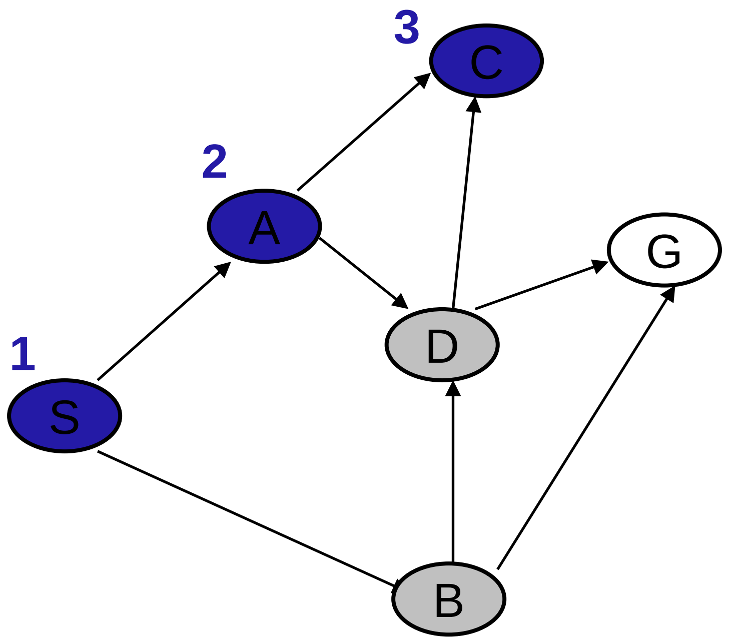
Also we have look at this as,



**Back up**

**or**

**backtrack**



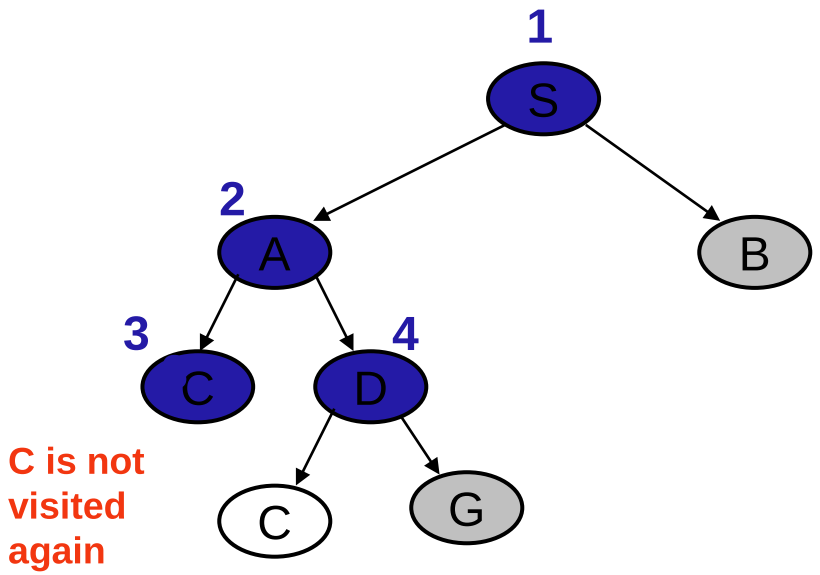
**Numbers indicate order pulled off of Q (expanded)**

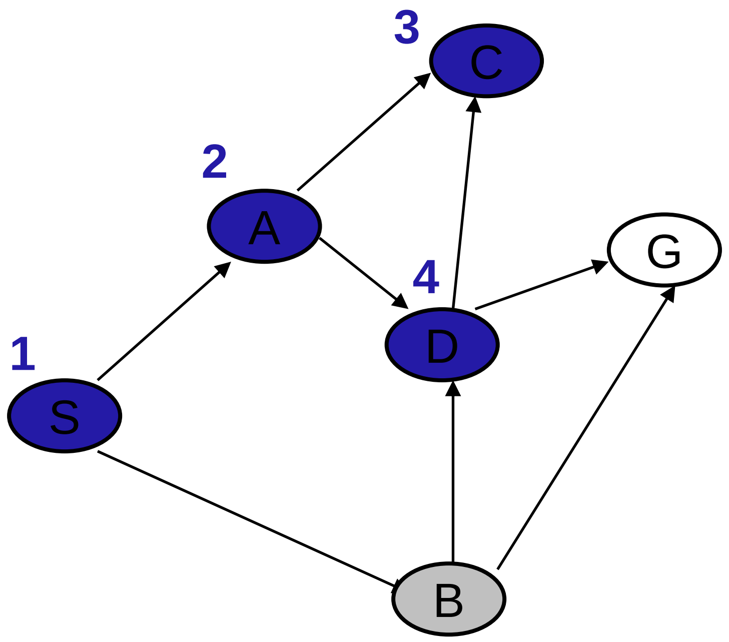
**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

## Depth First Search

Also we have look at this as,





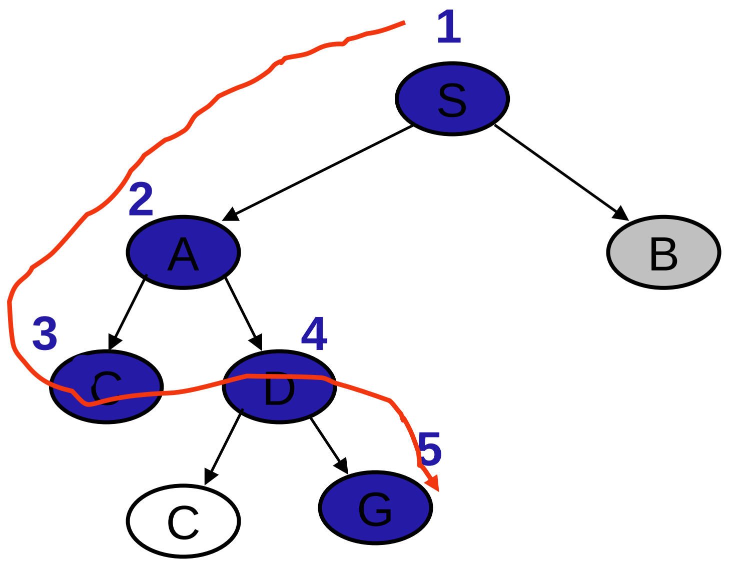
**Numbers indicate order pulled off of Q (expanded)**

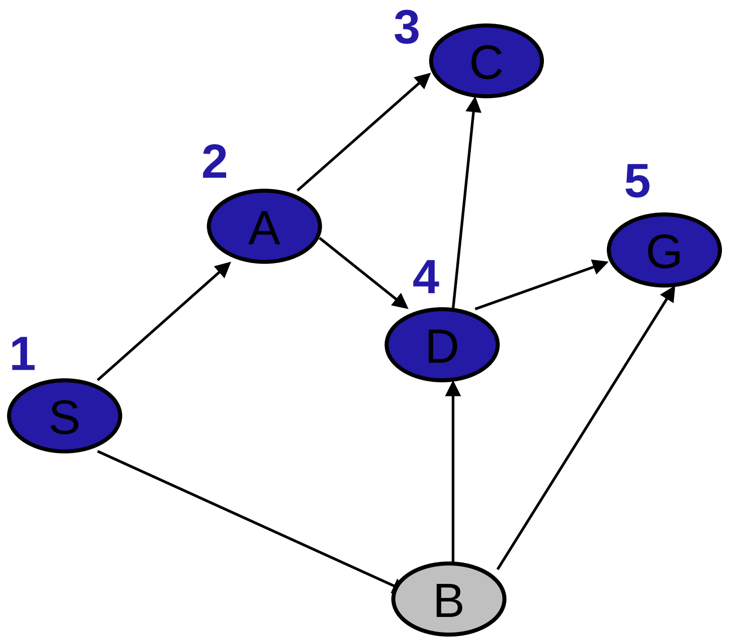
**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

# Depth First Search

Also we have look at this as,





**Numbers indicate order pulled off of Q (expanded)**

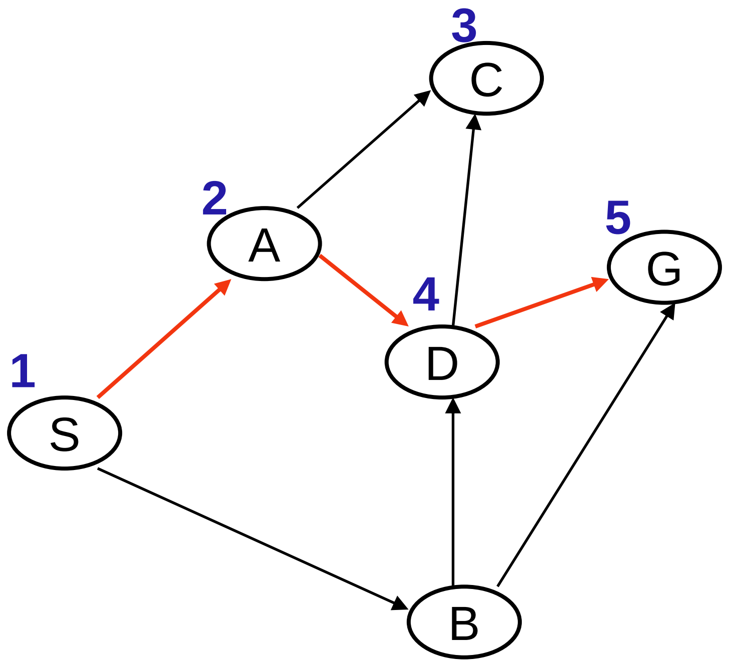
**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

# Depth First Search (without Visited List)

Pick First element of Q; Add path extensions to front of Q

| | Q |
|---|---|
| 1 | (S) |
| 2 | (A S) (B S) |
| 3 | (C A S) (D A S) (B S) |
| 4 | (D A S) (B S) |
| 5 | (C D A S) (G D A S) (B S) |
| 6 | $\boxed{\text{(G D A S)}}$ ( B S) |



Added paths in blue

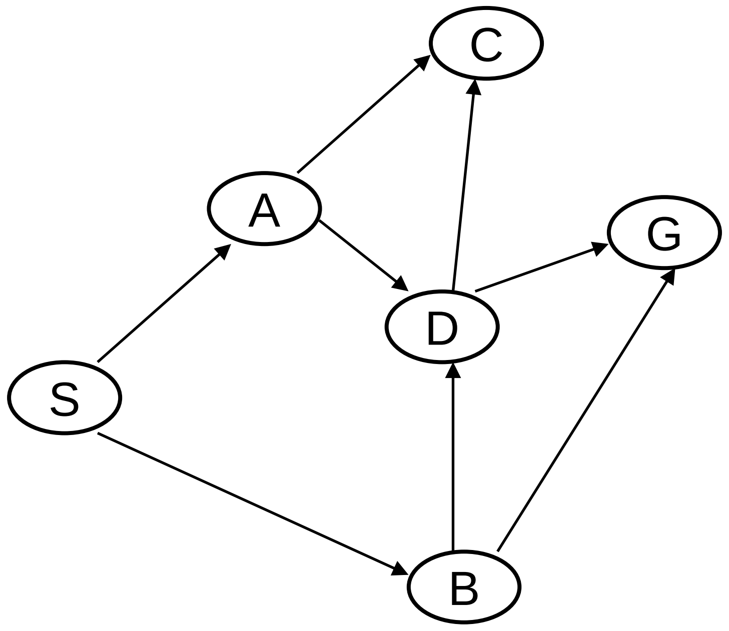
Path is shown in reversed order, the node's state is the first entry

Do not extend a path to a state if the resulting path would have a loop

## Breadth First Search

Pick First element of Q; Add path extensions to end of Q

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |



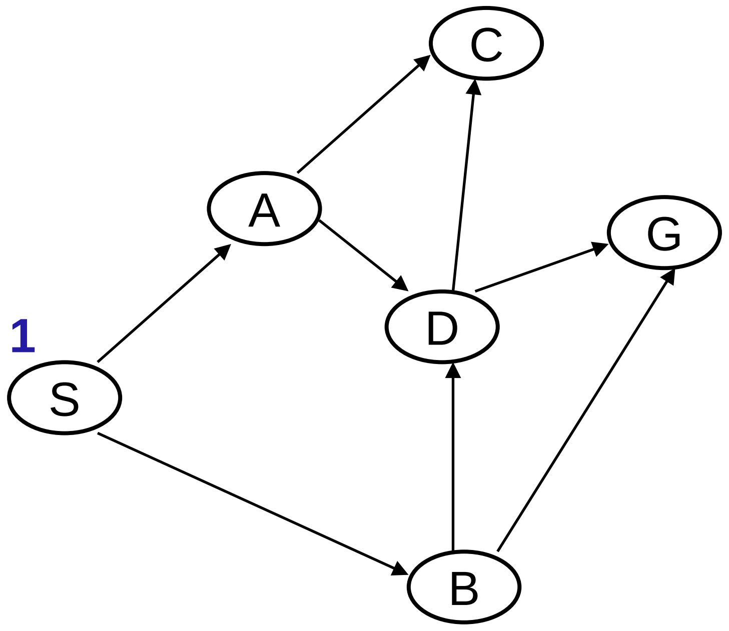
Added paths in blue

Path is shown in reversed order, the node's state is the first entry

## Breadth First Search

Pick First element of Q; Add path extensions to end of Q

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | | |
| 4 | | |
| 5 | | |



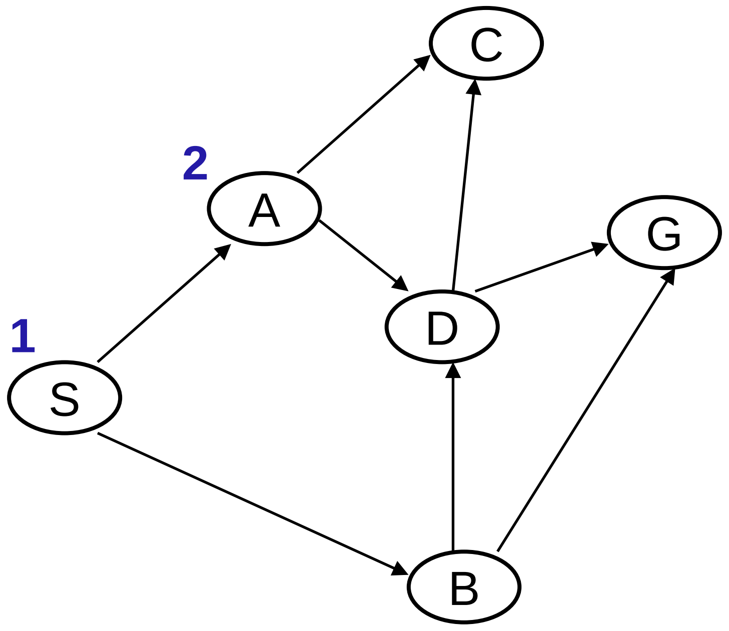
Added paths in blue

Path is shown in reversed order, the node's state is the first entry

# Breadth First Search

**Pick First element of Q; Add path extensions to end of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | **(B S) (C A S) (D A S)** | **C,D,B,A,S** |
| 4 | | |
| 5 | | |



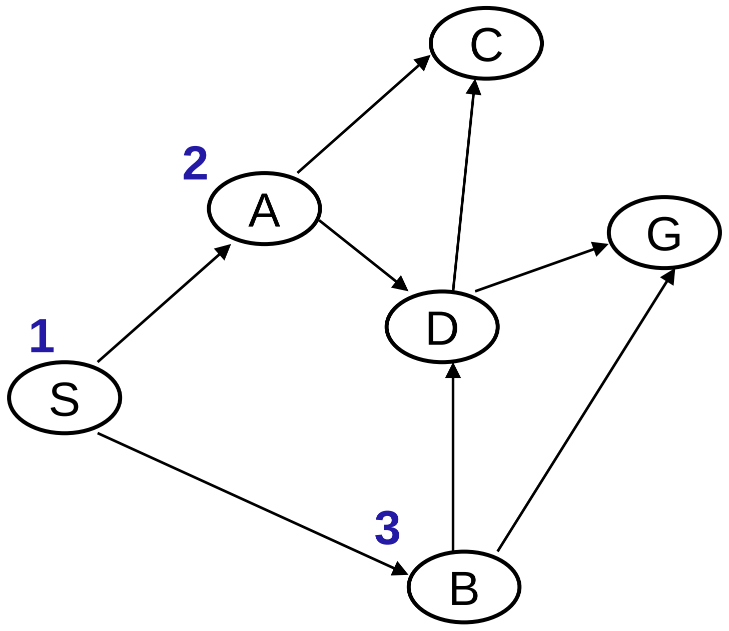
**Added paths in blue**

**Path is shown in reversed order, the node's state is the first entry**

## Breadth First Search

Pick First element of Q; Add path extensions to end of Q

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (B S) (C A S) (D A S) | C,D,B,A,S |
| 4 | (C A S) (D A S) (G B S) | G,C,D,B,A,S |
| 5 | | |
| 6 | | |



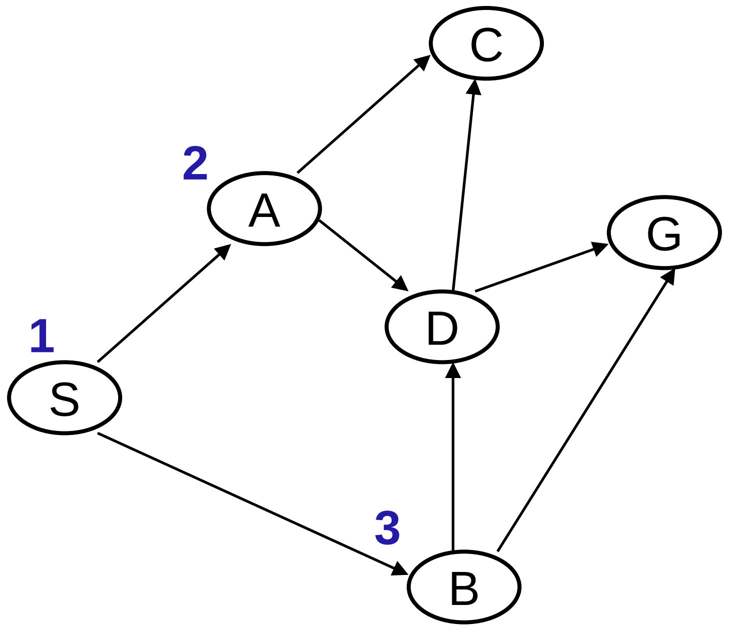
Added paths in blue

Path is shown in reversed order, the node's state is the first entry

## Breadth First Search

**Pick First element of Q; Add path extensions to end of Q**

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (B S) (C A S) (D A S) | C,D,B,A,S |
| 4 | (C A S) (D A S) (G B S)* | G,C,D,B,A,S |
| 5 | | |
| 6 | | |



**Added paths in blue**

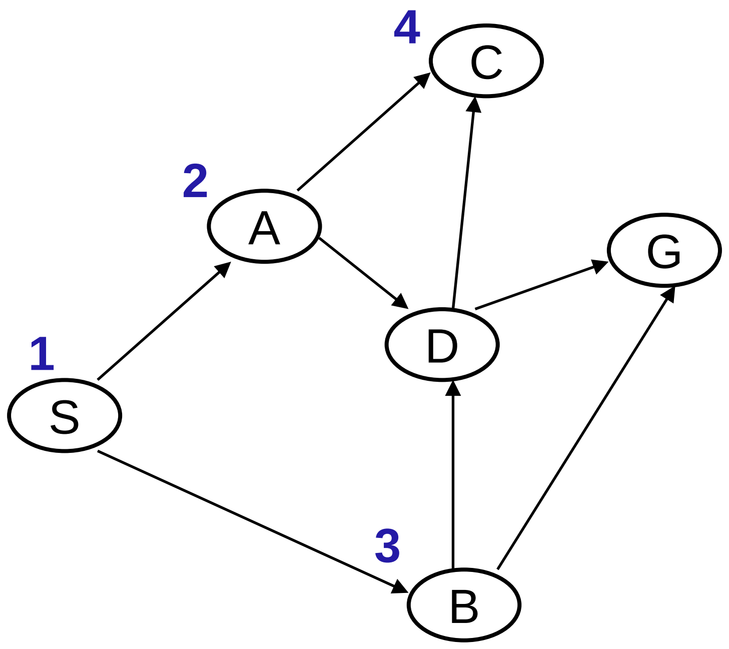
**Path is shown in reversed order, the node's state is the first entry**

***We could have stopped here, when the first path to the goal was generated**

# Breadth First Search

Pick First element of Q; Add path extensions to end of Q

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (B S) (C A S) (D A S) | C,D,B,A,S |
| 4 | (C A S) (D A S) (G B S)* | G,C,D,B,A,S |
| 5 | (D A S) (G B S) | G,C,D,B,A,S |
| 6 | | |



Added paths in blue

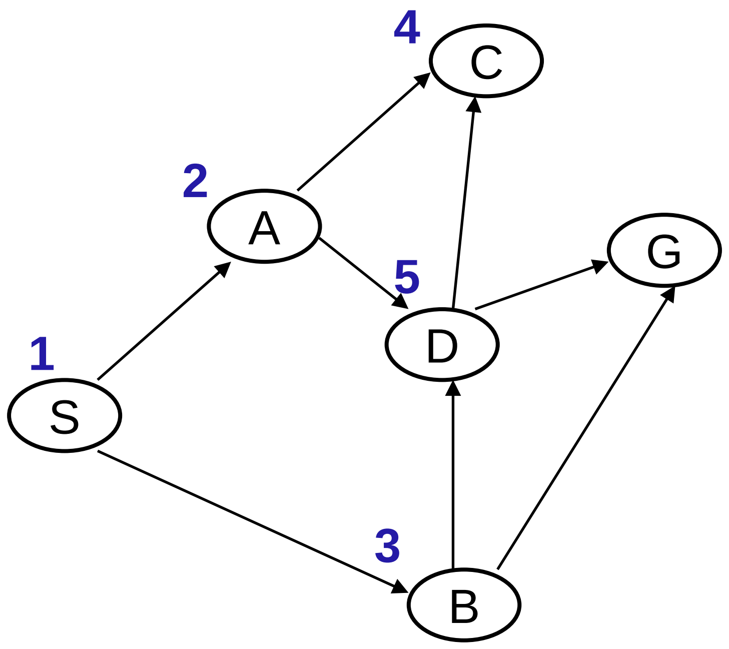
Path is shown in reversed order, the node's state is the first entry

*We could have stopped here, when the first path to the goal was generated

# Breadth First Search

**Pick First element of Q; Add path extensions to end of Q**

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (B S) (C A S) (D A S) | C,D,B,A,S |
| 4 | (C A S) (D A S) (G B S)* | G,C,D,B,A,S |
| 5 | (D A S) (G B S) | G,C,D,B,A,S |
| 6 | | |



**Added paths in blue**

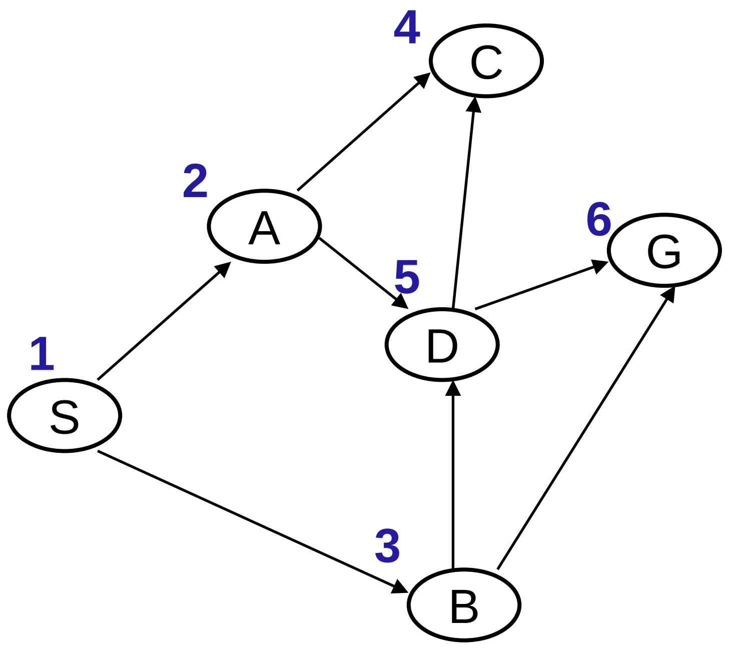
**Path is shown in reversed order, the node's state is the first entry**

***We could have stopped here, when the first path to the goal was generated**

# Breadth First Search

**Pick First element of Q; Add path extensions to end of Q**

| | Q | Visited |
|---|---|---|
| 1 | (S) | S |
| 2 | (A S) (B S) | A, B, S |
| 3 | (B S) (C A S) (D A S) | C,D,B,A,S |
| 4 | (C A S) (D A S) (G B S)* | G,C,D,B,A,S |
| 5 | (D A S) (G B S) | G,C,D,B,A,S |
| 6 | (G B S) | G,C,D,B,A,S |



**Added paths in blue**

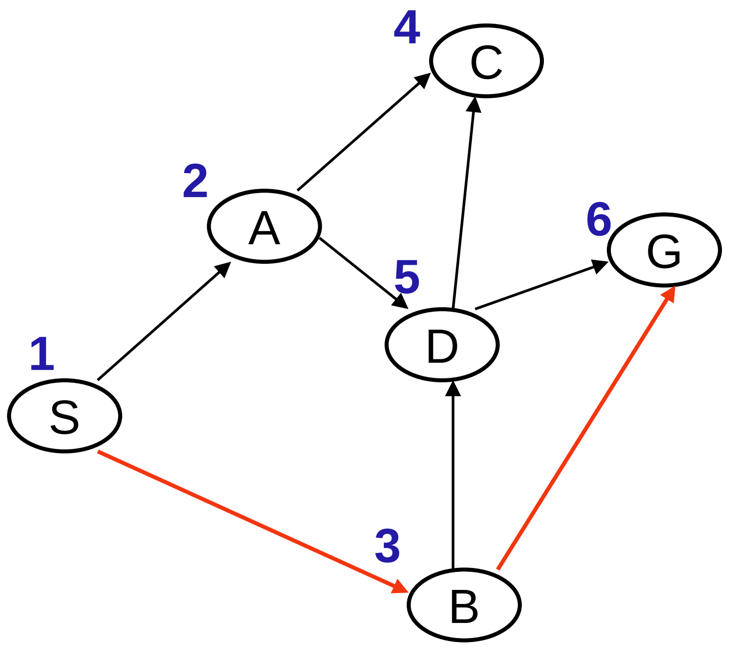
**Path is shown in reversed order, the node's state is the first entry**

***We could have stopped here, when the first path to the goal was generated**

# Breadth First Search

**Pick First element of Q; Add path extensions to end of Q**

| | **Q** | **Visited** |
|---|---|---|
| 1 | **(S)** | **S** |
| 2 | **(A S) (B S)** | **A, B, S** |
| 3 | **(B S) (C A S) (D A S)** | **C,D,B,A,S** |
| 4 | **(C A S) (D A S) (G B S)*** | **G,C,D,B,A,S** |
| 5 | **(D A S) (G B S)** | **G,C,D,B,A,S** |
| 6 | **(G B S)** | **G,C,D,B,A,S** |



**Added paths in blue**

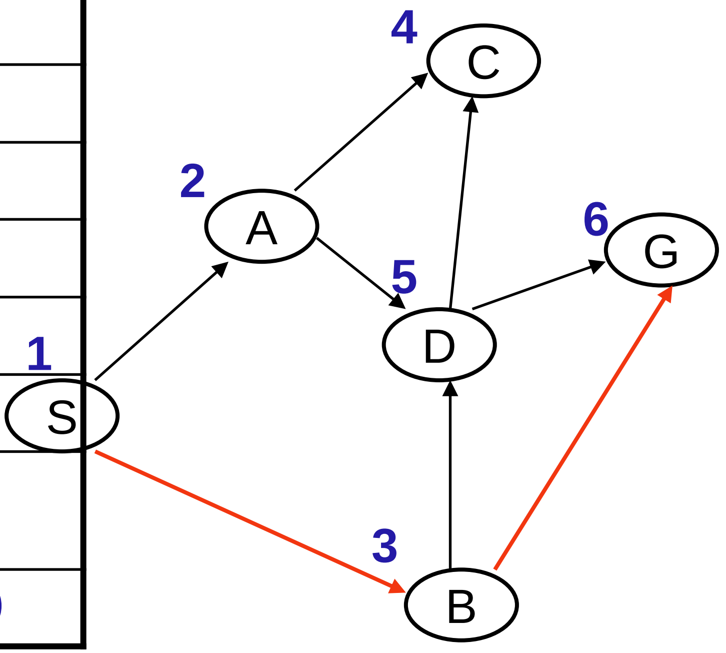
**Path is shown in reversed order, the node's state is the first entry**

***We could have stopped here, when the first path to the goal was generated**

# Breadth First (without Visited list)

Pick First element of Q; Add path extensions to end of Q

| | Q |
|---|---|
| 1 | (S) |
| 2 | (A S) (B S) |
| 3 | (B S) (C A S) (D A S) |
| 4 | (C A S) (D A S) (D B S) (G B S)* |
| 5 | (D A S) (D B S) (G B S) |
| 6 | (D B S) (G B S) ( C D A S) (G D A S) |
| 7 | (G B S) (C D A S) (G D A S) (C D B S) (G D B S) |

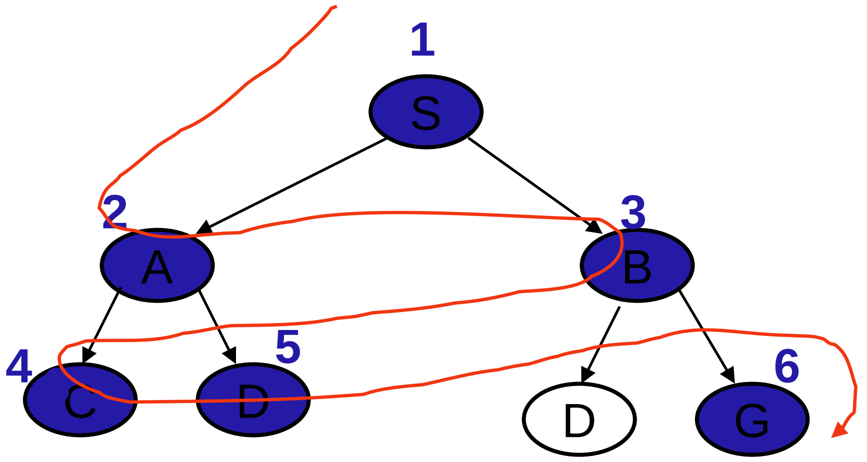


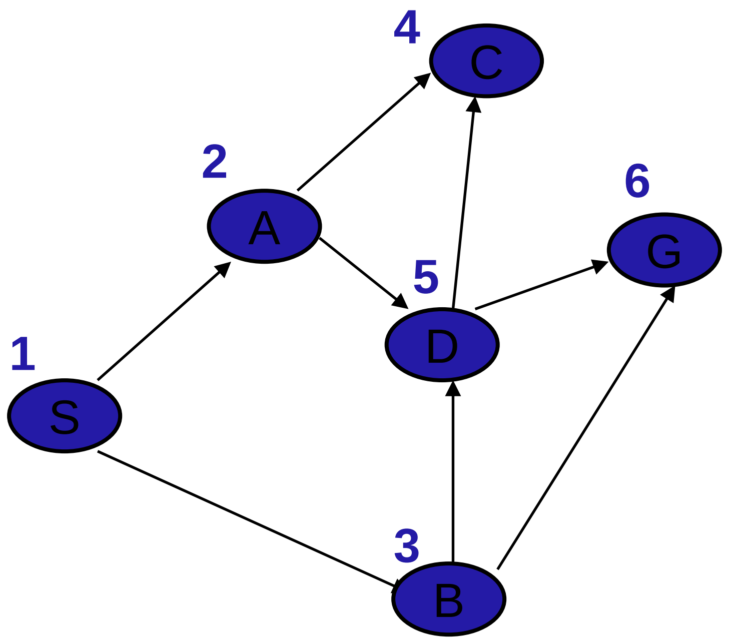
Added paths in blue

Path is shown in reversed order, the node's state is the first entry

## Breadth First Search

Also we have look at this as,





D is not visited again

**Numbers indicate order pulled off of Q (expanded)**

**Dark blue fill = Visited & Expanded**

**Light grey fill = Visited**

## Implementing Issues: Finding the best node

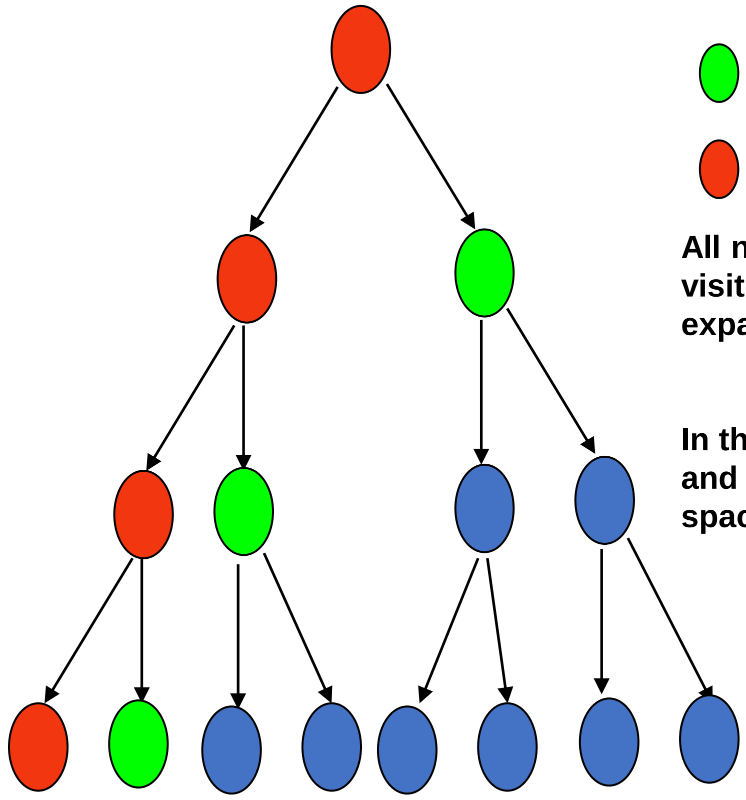
There are many possible approaches to finding the best node in Q.

- Scanning Q to find lowest values
- Sorting Q and picking the first element
- Keeping the Q sorted by doing “sorted” insertions
- Keeping Q as a priority queue – like in data structures

Which of these is best will depend among other thing on how many children nodes have on average

# Worst Case Space

**Max Q size = Max(#Visited - #Expanded)**

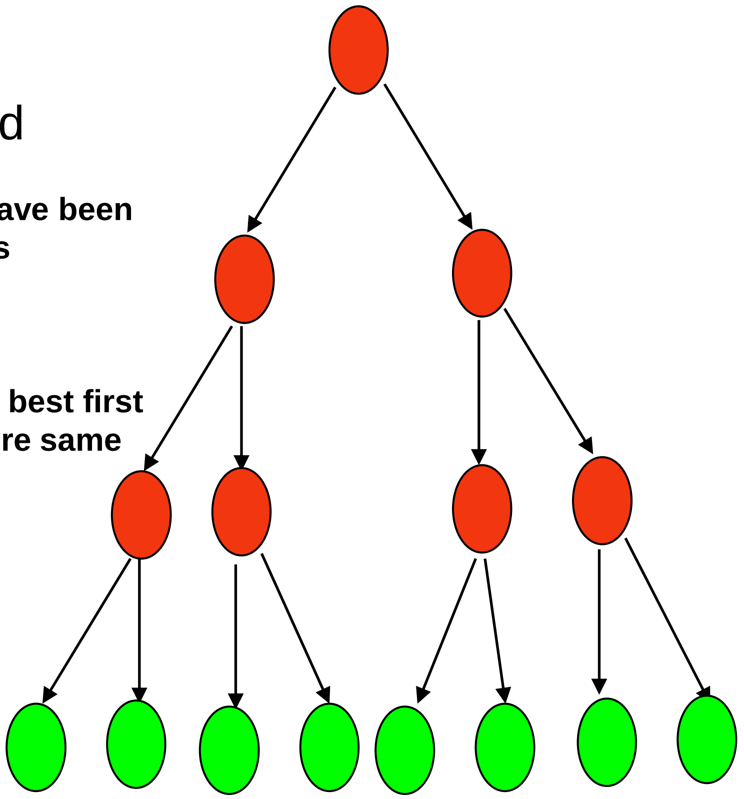


🟢 visited

🔴 expanded

**All nodes at depth d have been visited but none of it is expanded**

**In the worst case both best first and breadth first require same space**



**Depth First max Q size**

**$(b-1)d \approx bd$**

**Breadth First max Q size**

**$b^d$**

# Worst Case Space

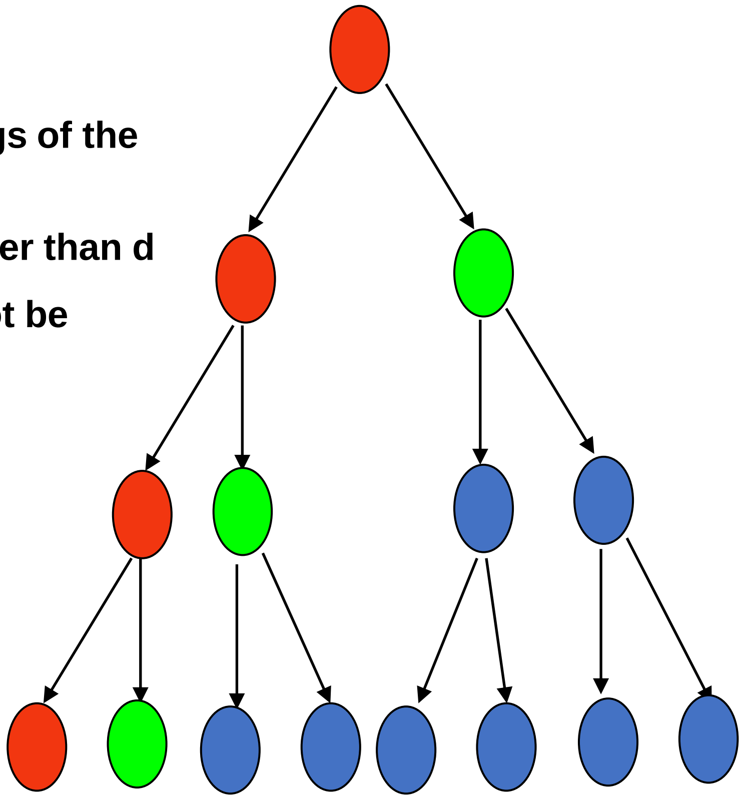
**Max Q size = Max(#Visited - #Expanded)**

**In a tree structured search**

**Q holds unexpanded siblings of the node**

**Path length cannot be greater than d**

**Unexpanded siblings cannot be greater than b-1**



| | |
|---|---|
| 🟢 | visited |
| 🔴 | expanded |

**Depth First max Q size**

**$(b-1)d \approx bd$**

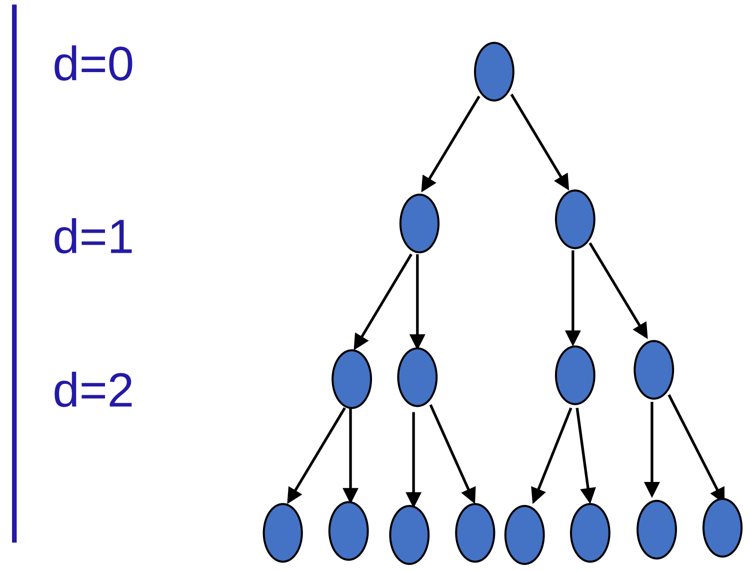
# Worst Case Running Time

**Max time ∞ Max #Visited**

**The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game**

**All the searches, with or without visited list, may have to visit each state at least once, in the worst case**

**So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the "depth" parameter.**



b=2

**Or is Uniform**

**Number of choices in a move**

**d is depth**

**b is branching factor**

$b^d < (b^{d+1} - 1)/(b - 1) < b^{d+1}$

**states in tree**

## Cost and Performance of Any-Path Methods

**Searching a tree with branching factor b and depth d**
**(without using a Visited list – tree search)**

| Search Method | Worst Time | Worst Space | Fewest States? | Guaranteed to find Path? |
|---|---|---|---|---|
| **Depth-first** | $b^{d+1}$ | bd | No | Yes* |
| **Breadth-first** | $b^{d+1}$ | $b^d$ | Yes | Yes |
| **Best-first** | $b^{d+1**}$ | $b^d$ | No | Yes* |

**If there are no infinitely long paths in the search space*

***Best-first needs more time to locate the best node in Q*

*Not very precise estimates but just tradeoffs*

**Worst case time is proportional to number of nodes added to Q**

**Worst case space is proportional to maximal length of Q**

**DFS requires less space i.e. linear**

## Cost and Performance of Any-Path Methods

**Searching a tree with branching factor b and depth d**
**(using a Visited list)**

| **Search Method** | **Worst Time** | **Worst Space** | **Fewest States?** | **Guaranteed to find Path?** |
|---|---|---|---|---|
| **Depth-first** | $b^{d+1}$ | ~~bd~~ $b^{d+1}$ | No | Yes* |
| **Breadth-first** | $b^{d+1}$ | ~~$b^d$~~ $b^{d+1}$ | Yes | Yes |
| **Best-first** | $b^{d+1}$** | ~~$b^d$~~ $b^{d+1}$ | No | Yes* |

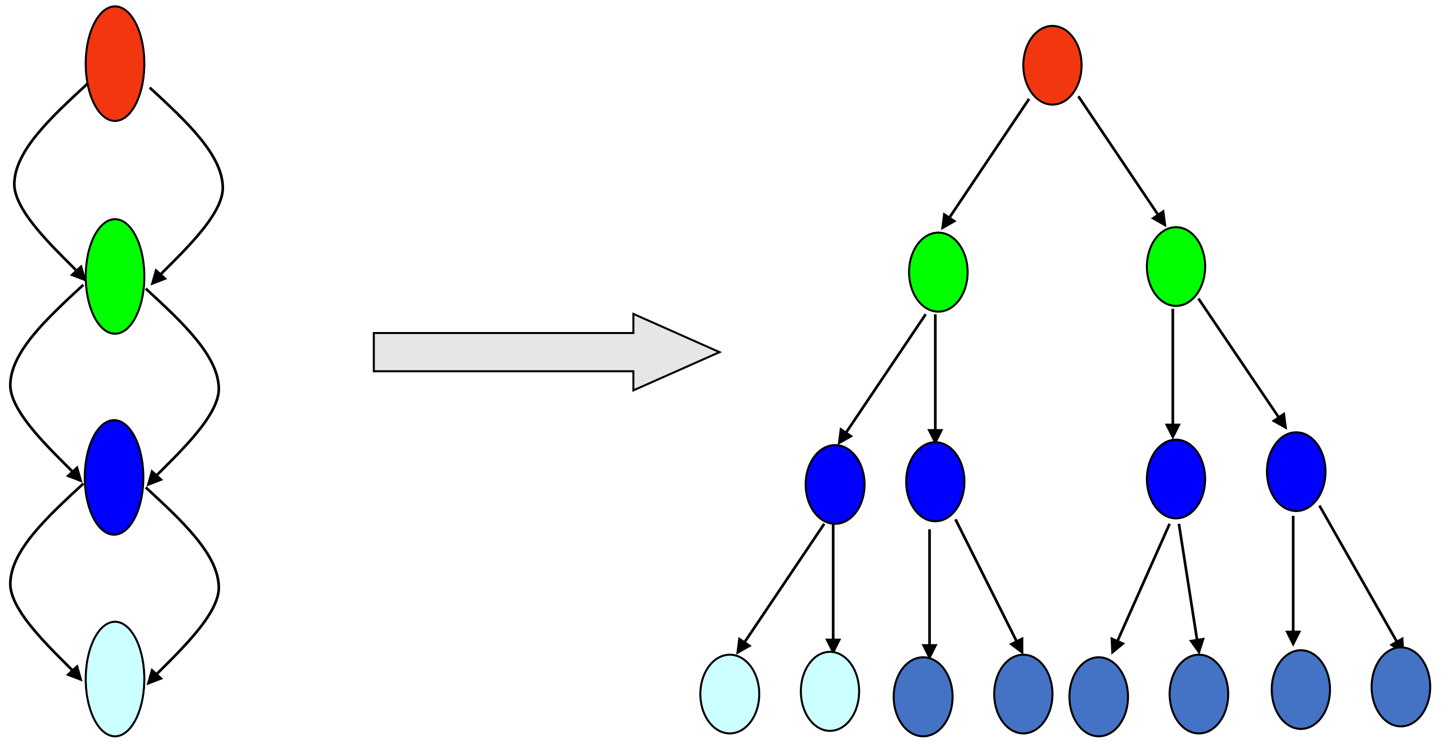
**If there are no infinitely long paths in the search space*

***Best-first needs more time to locate the best node in Q*

**Worst case time is proportional to number of nodes added to Q**

**Worst case space is proportional to maximal length of Q (and Visited list)**

## States Vs Paths



Visited list can save lot of time as shown by this example.....there could be exponentially larger number of path than states

## Space

**In large search problem, memory is often the limiting factor**

**Imagine search a tree with branching factor 8 and depth 10. Assume a node requires just 8 bytes of storage. Then breadth-first search might require up to**

?

$$(2^3)^{10} \times 2^3 = 2^{33} \text{ bytes} = 8{,}000 \text{ Mbytes} = 8\text{Gbytes}$$

**One strategy is to trade time for memory. For example, we can emulate breadth-first search by repeated applications of depth-first, each up to a preset depth limit. This is called progressive deepening search (PDS)**

1- **C=1**
2- **Do DFS to max depth C. If path found, return it.**
3- **Otherwise, increment C and go to 2**

## PDS – Best of Both Worlds

**Depth-First Search (DFS) has small space requirements (linear in depth) but has major problems:**

- **DFS can run forever in search spaces with infinite length paths**
- **DFS does not guarantee finding shallowest goal**

**Breadth first Search (BSF) guarantees finding shallowest goal, even in the presence of infinite paths, but it has one great problem:**

- **BFS requires a great deal of space (exponential in depth)**

**Progressive Deepending Search (PDS) has the advantages of DFS and BFS**

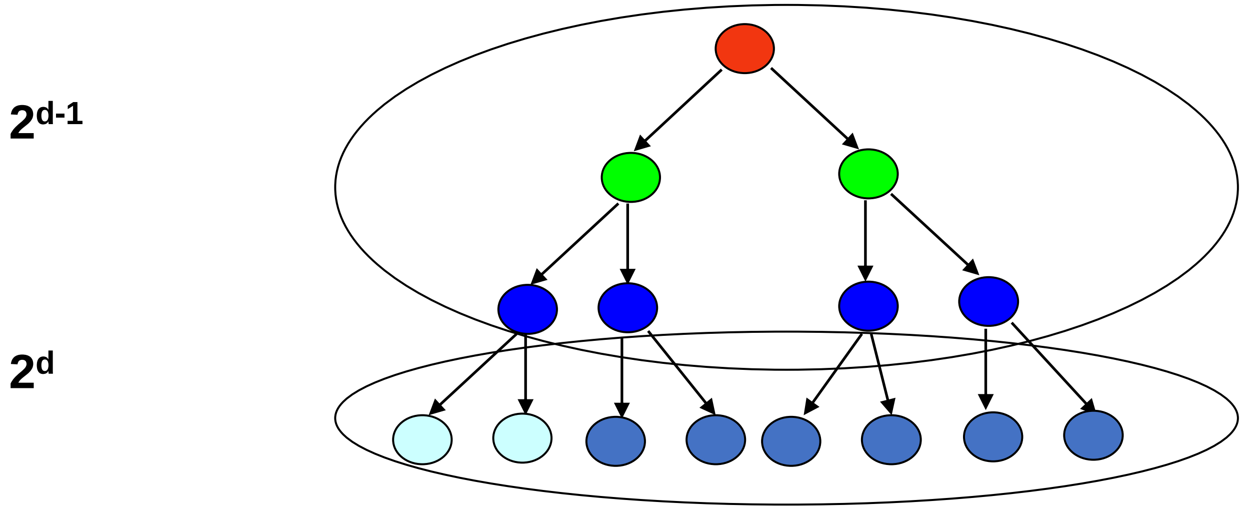
- **PDS has small space requirements (linear in depth)**
- **PDS guarantees finding shallowest goal**

# Progressive Deepening Search

**Isn't PDS too expensive?** **You look at nodes again and again so wastage of time but……only in small searches or graphs**

**In exponential trees, time is dominated by deepest search**

**For example, if branching factor is 2, then the number of nodes at depth d is $2^d$ while the total number of nodes in all previous levels is $2^{d-1}$, so the difference between looking at whole tree versus only the deepest level is at worst a factor of 2 in performance**



## Progressive Deepening Search

**Compare the ratio of average time spent on PDS with average time spent on a single DFS with the full depth tree:**

**(Avg time for PDS)/(Avg time for DFS) ≈ (b+1)/(b-1)**

| b | ratio |
|---|---|
| 2 | 3 |
| 3 | 2 |
| 5 | 1.5 |
| 25 | 1.08 |
| 100 | 1.02 |

**Progressive deepening is an effective strategy for difficult searches.**

## Questions

?