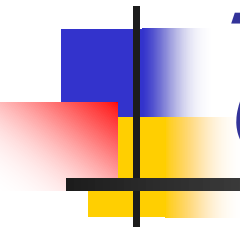
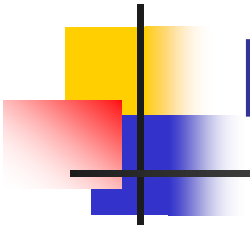


Introduction to Artificial Intelligence (G51IAI)

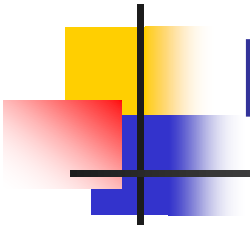
Dr Rong Qu

Blind Searches



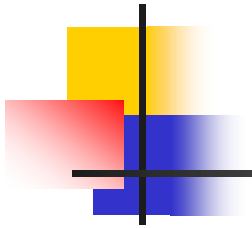
Blind Searches

```
Function GENERAL-SEARCH (problem, QUEUING-FN) returns a solution or failure
    nodes = MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    Loop do
        If nodes is empty then return failure
        node = REMOVE-FRONT(nodes)
        If GOAL-TEST[problem] applied to STATE(node) succeeds
            then return node
        nodes = QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    End Loop
End Function
```

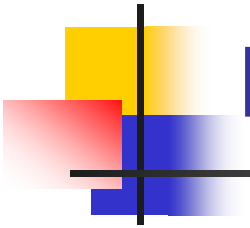


Blind Searches

- The difference between searches lies in the order in which nodes are selected for expansion
- The search always visits the first node in the fringe queue
  - REMOVE
- The only way to control the ordering
  - INSERT



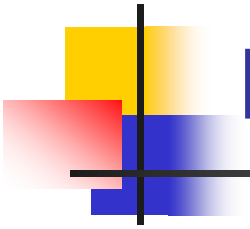
Breadth First Search



Breadth First Search - Method

- Expand Root Node First
- Expand all nodes at level 1 before expanding level 2

  OR
- Expand all nodes at level *d* before expanding nodes at level *d*+1



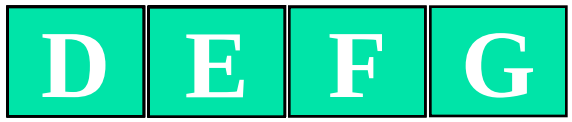
Breadth First Search - Implementation

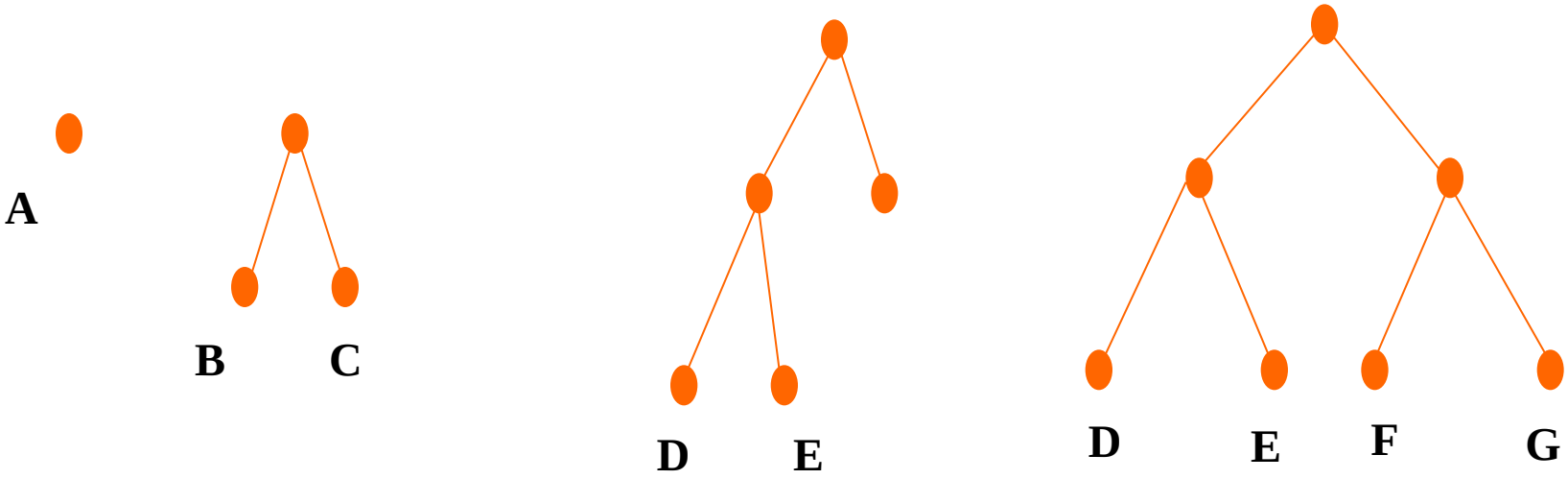
- Use a queuing function that adds nodes to the end of the queue

Function BREADTH-FIRST-SEARCH(*problem*) returns a solution or failure

Return GENERAL-SEARCH (*problem*, ENQUEUE-AT-END)

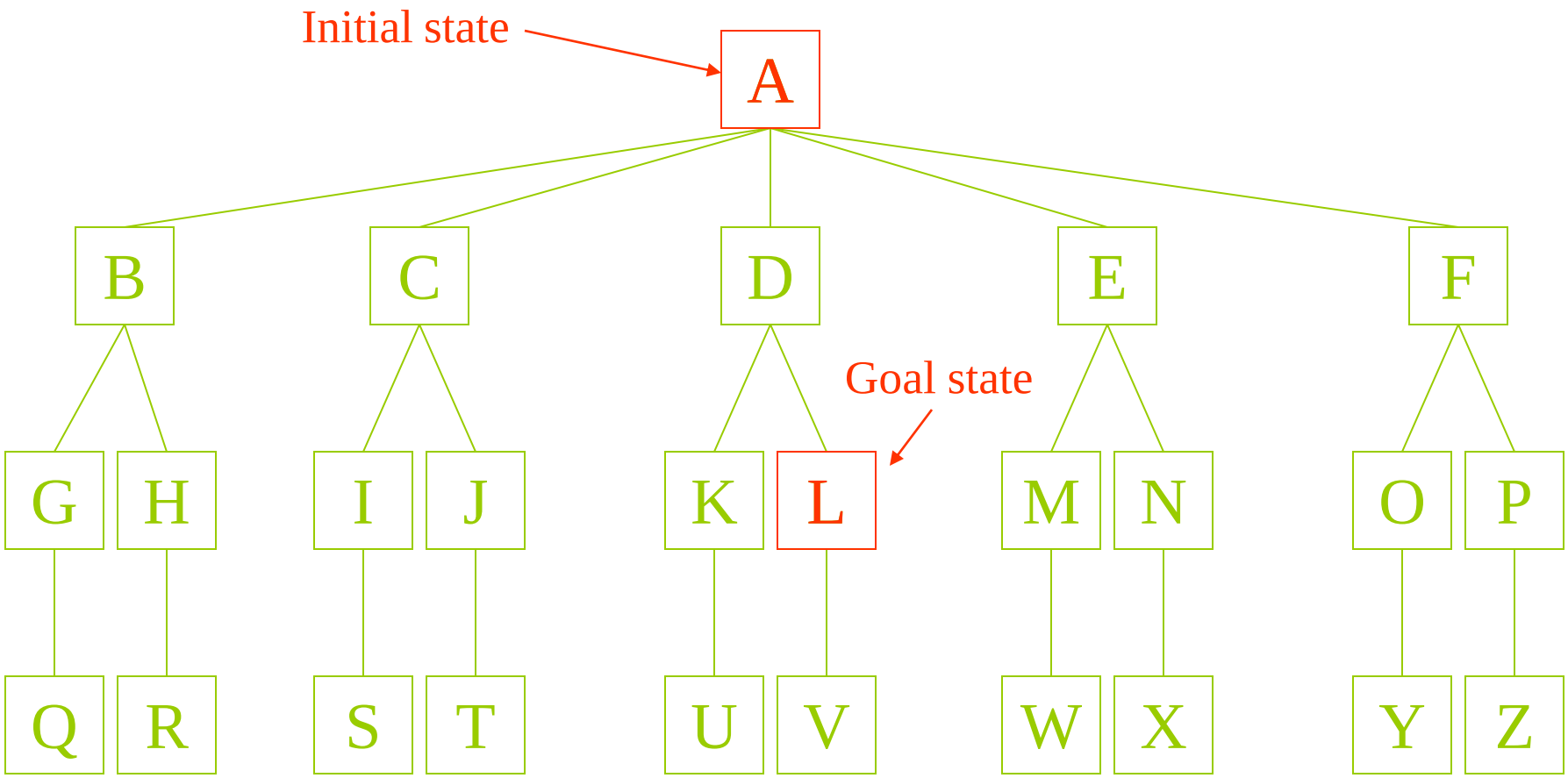
Breadth First Search - Implementation



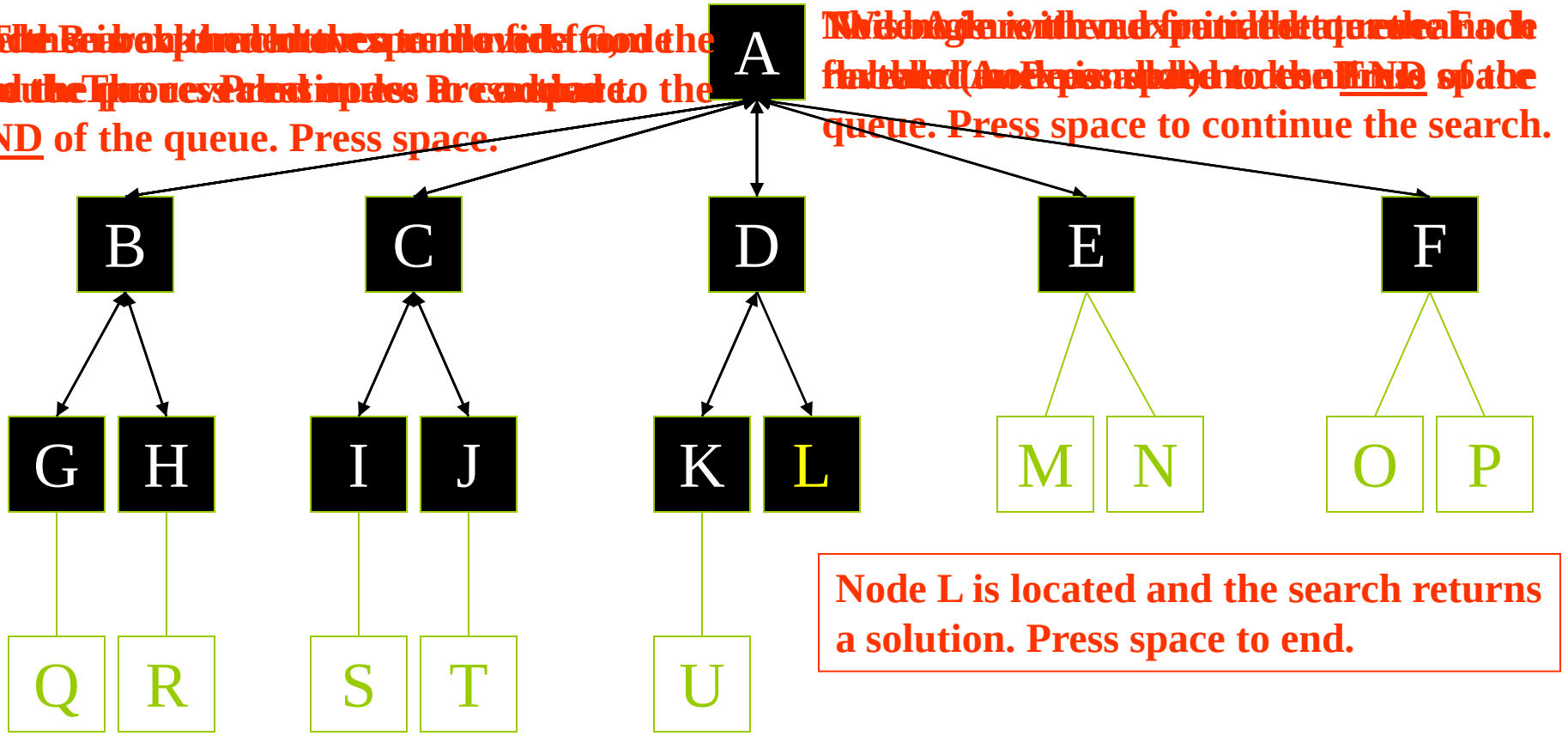


node = REMOVE-FRONT(nodes)
If GOAL-TEST[problem] applied to STATE(node) succeeds
    then return node
nodes = QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))

The example node set



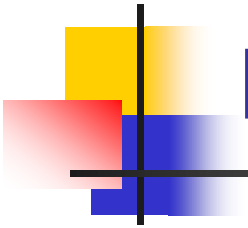
Press space to see a BFS of the example node set



Press space to begin the search

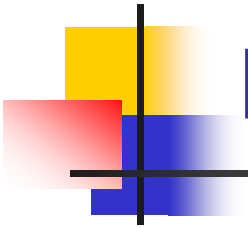
Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 11	**FINISHED SEARCH**	**Current level: 2**

BREADTH-FIRST SEARCH PATTERN



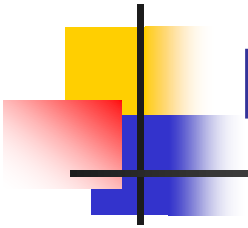
Evaluating Breadth First Search

- Observations
  - Very systematic
  - If there is a solution breadth first search is guaranteed to find it
  - If there are several solutions then breadth first search will always find the shallowest goal state first and if the cost of a solution is a non-decreasing function of the depth then it will always find the cheapest solution
    - Path cost



Evaluating Breadth First Search

- Evaluating against four criteria
  - Optimal
  - Complete
  - Time complexity
  - Space complexity



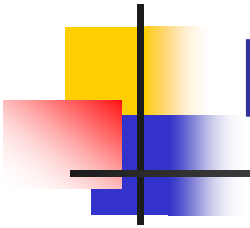
Evaluating Breadth First Search

- Evaluating against four criteria
  - Complete?
    - Yes
  - Optimal?
    - Yes

Evaluating Breadth First Search

- Evaluating against four criteria
  - Space Complexity
    - $O(b^d)$
  - Time Complexity
    - $O(b^d)$ i.e. number of leaves
    - $1 + b + b^2 + b^3 + \ldots + b^{d-1}$ i.e. $O(b^d)$
    - b: the average branching factor
    - d: is the depth of the search tree

Note : The space/time complexity could be less as the solution may be found somewhere before the d^{th} level (depends on the problem).



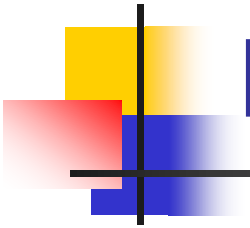
Exponential Growth

- Exponential growth quickly makes complete state space searches unrealistic
- If the branching factor was 10, by level 5 we would need to search 100,000 nodes, i.e. 10^5

Exponential Growth

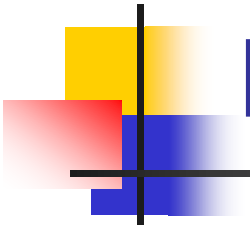
Depth	Nodes	Time		Memory	
0	1	1	millisecond	100	bytes
2	111	0.1	second	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

Time and memory requirements for breadth-first search, assuming a branching factor of 10, 100 bytes per node and searching 1000 nodes/second



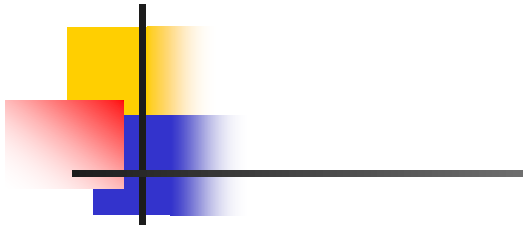
Breadth First Search - Observations

- Space is more of a factor to breadth first search than time
- Time is still an issue
  - Who has 35 years to wait for an answer to a level 12 problem (or even 128 days to a level 10 problem)

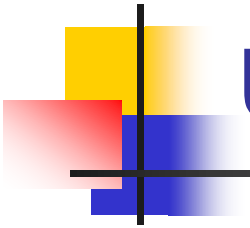


Breadth First Search - Observations

- It could be argued that as technology gets faster then exponential growth will not be a problem
- But even if technology is 100 times faster
  - we would still have to wait 35 years for a level 14 problem and what if we hit a level 15 problem!

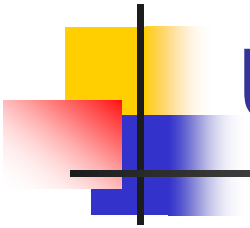


Uniform Cost Search (vs. BFS)



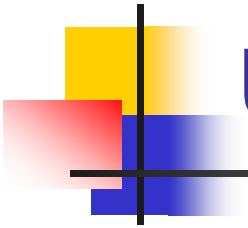
Uniform Cost Search (vs. BFS)

- BFS will find the optimal (shallowest) solution as long as the cost is a function of the depth
- Suppose that we have a tree in which all the weights of branches are one
- Weight of a path from the root to a node N is just the depth of node N



Uniform Cost Search (vs. BFS)

- Uniform Cost Search can be used when this is not the case
  - will find the cheapest solution provided *that the cost of the path never decreases as we proceed along the path*
- Uniform Cost Search works by expanding the lowest cost node on the fringe



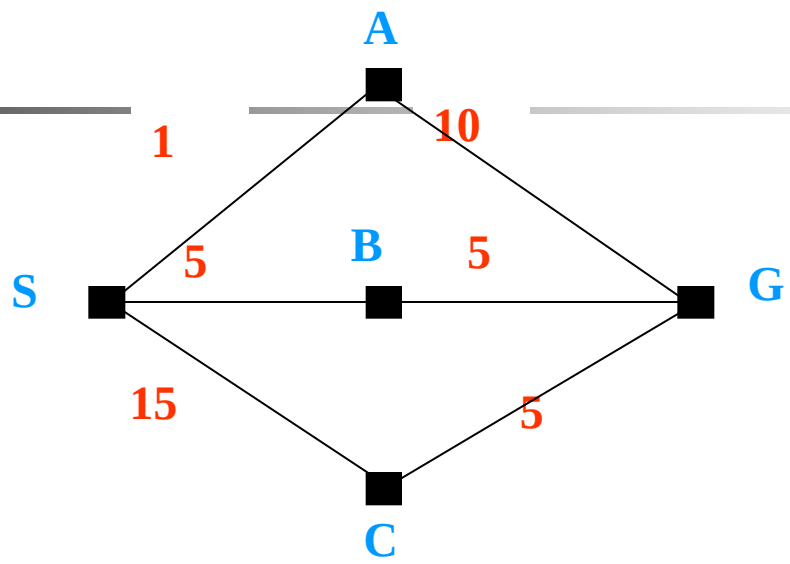
Uniform Cost Search (vs. BFS)

- Cost of a node *n*
  - the total cost of the path from the root to *n*
- "Search all nodes of cost *c* before those of cost *c*+1"
  - In BFS deeper nodes always arrive after shallower nodes
  - In UCS the costs of new nodes do not have such a nice pattern

Uniform Cost Search - implementation

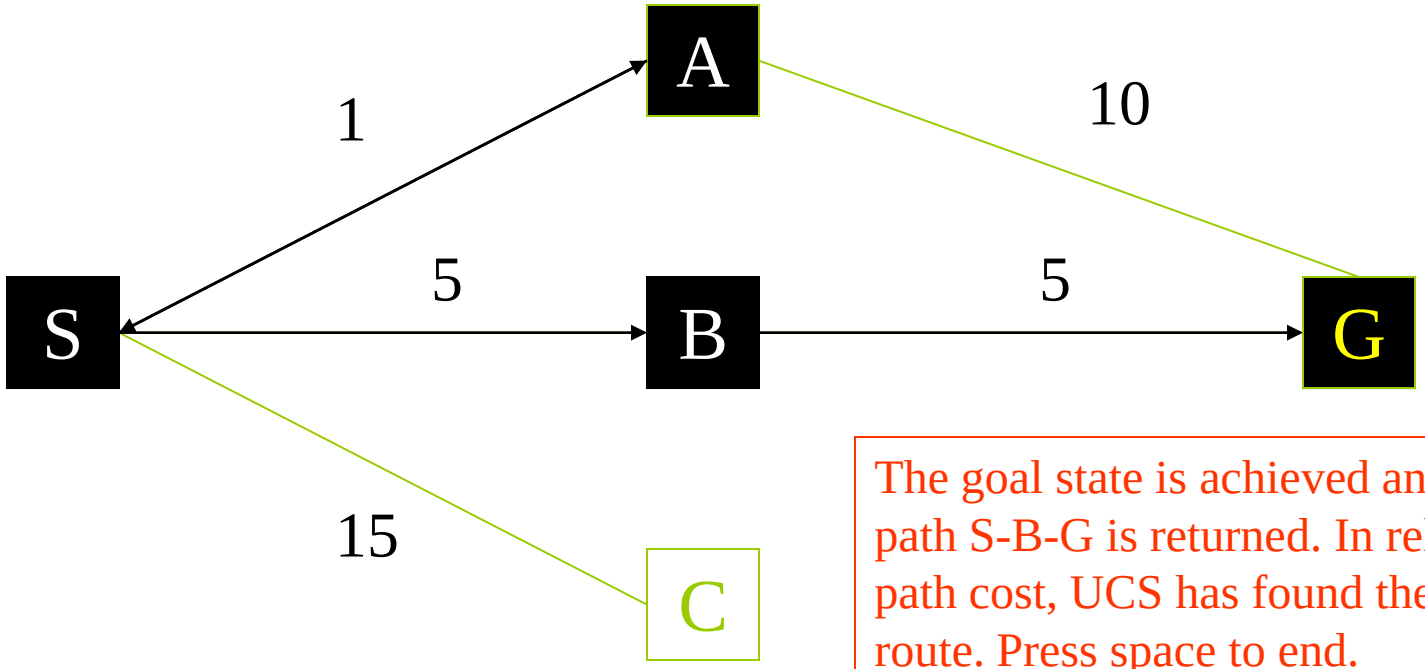
- In UCS we need to
  1. explicitly store the cost g of a node
  2. explicitly use such costs in deciding the ordering in the queue

- Always remove the smallest cost node first
  - sort the queue in increasing order
  - alternatively, search the queue and remove the smallest cost
  - Nodes removed by cost, not by order of arrival



- BFS will find the path SAG, with a cost of 11, but SBG is cheaper with a cost of 10
- UCS will find the cheaper solution (SBG). It will find SAG but will not see it as it is not at the head of the queue

proceed to goal state. Press space.

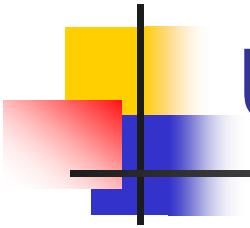


The goal state is achieved and the path S-B-G is returned. In relation to path cost, UCS has found the optimal route. Press space to end.

Press space to begin the search

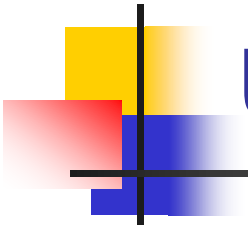
Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 3	**FINISHED SEARCH**	**Current level: 2**

UNIFORM COST SEARCH PATTERN



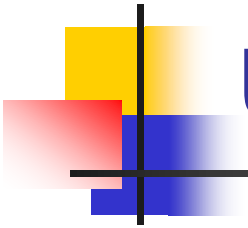
Uniform Cost Search - properties

- Completeness:
  - If there is a path to a goal then UCS will find it
  - If there is no path, then UCS will eventually report that the goal is unreachable
- Optimality:
  - UCS will report a minimum cost path (there might be many)



UCS vs. BFS - queue

- Breadth First Search
  - Nodes added to the end of the queue
- Uniform Cost Search
  - Nodes ordered by their cost
- Queue
  - Open nodes/fringe nodes
  - First node removed/expended during the search



UCS vs. BFS – complete & optimal

- Breadth First Search
  - Optimal
    - Only if the branch cost is the same
- Uniform Cost Search
  - Optimal
    - Even if the branch cost is different
- Complete
  - Systematic search throughout the whole tree

UCS vs. BFS – complexity

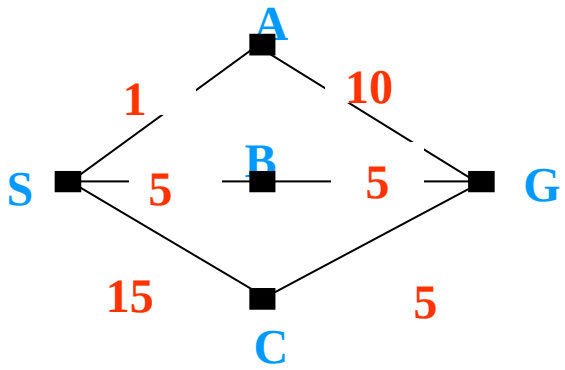
- Time and space complexity
  $O(b^d)$ (bounded by b^d)
- UCS is usually better than BFS

- UCS = BFS
  - When all solutions rather than just one solution is needed
  - When all branches have the same cost

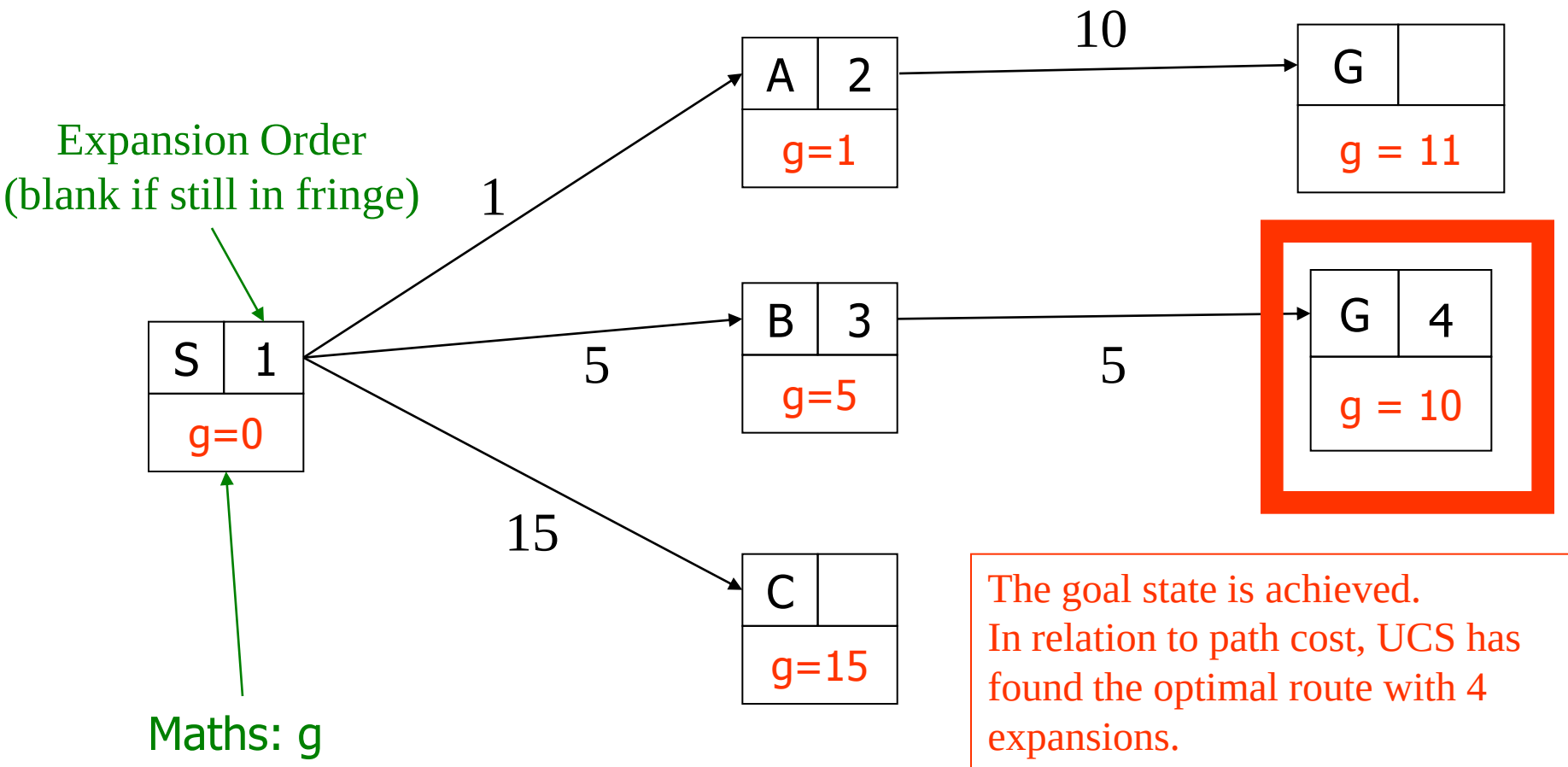
  - We are talking about the worst case scenario

Uniform Cost Search - implementation

Node Name	Expansion Number (order of expansion) Blank if in fringe
Any needed maths. Usually: values of g, h, and/or f	



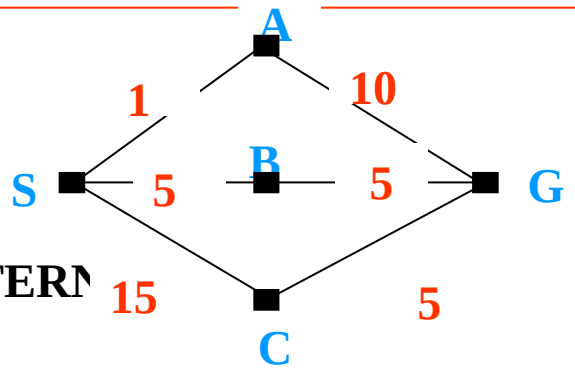
In terms of a search tree we could represent the UCS example as follows ....



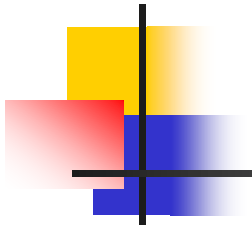
The goal state is achieved.
In relation to path cost, UCS has found the optimal route with 4 expansions.

Note that G is split into two nodes corresponding to the two paths.

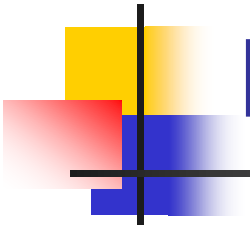
Press space to begin the search



UNIFORM COST SEARCH PATTERN



Depth First Search



Depth First Search - Method

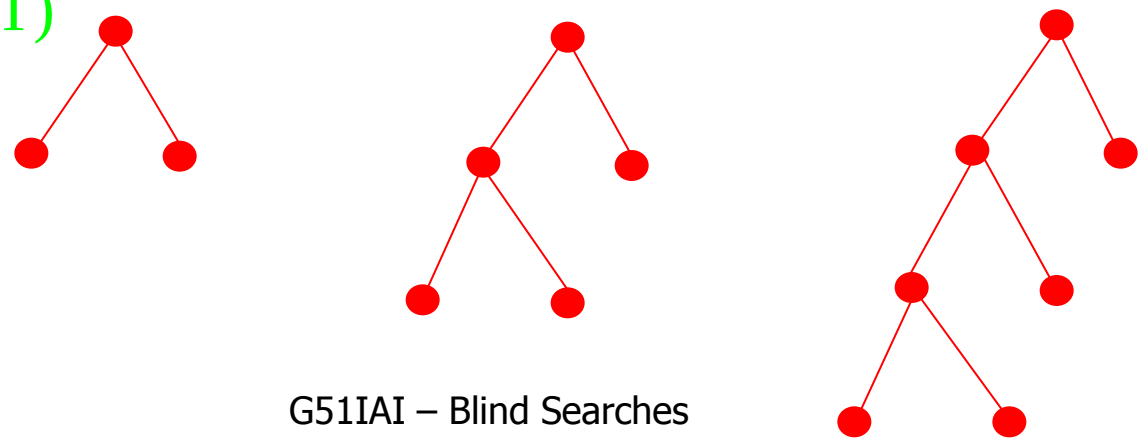
- Expand Root Node First
- Explore one branch of the tree before exploring another branch

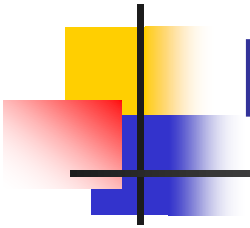
Depth First Search - Implementation

- Use a queuing function that adds nodes to the front of the queue

Function DEPTH-FIRST-SEARCH(problem) returns a solution or failure

Return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)



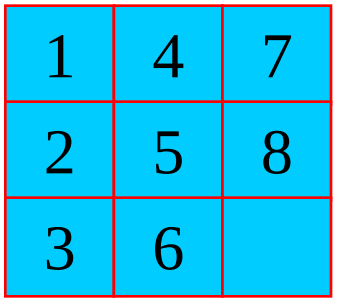


Depth First Search - Observations

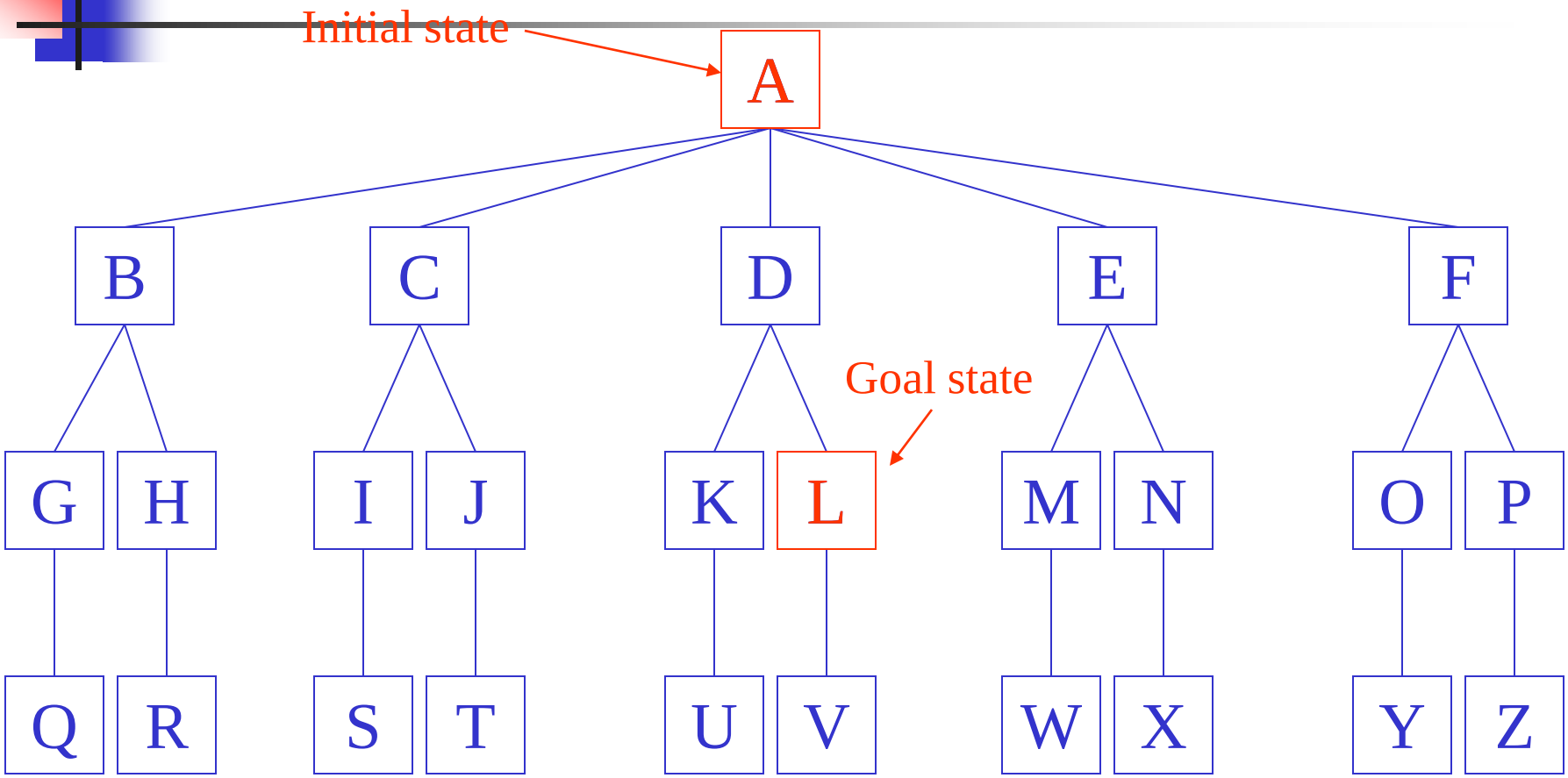
- Space complexity
  - Only needs to store the path from the root to the leaf node as well as the unexpanded nodes
  - For a state space with a branching factor of b and a maximum depth of m, DFS requires storage of bm nodes
- Time complexity
  - b^m in the worst case

Depth First Search - Observations

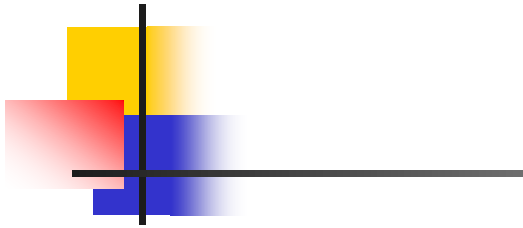
- If DFS goes down a infinite branch it will not terminate if it does not find a goal state
- If it does find a solution there may be a better solution at a lower level in the tree
- Therefore, depth first search is neither complete nor optimal



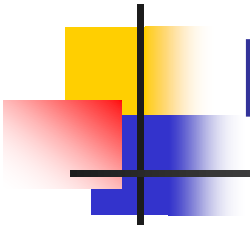
The example node set



Press space to see a BFS of the example node set

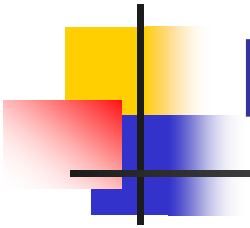


Depth Limited Search (vs. DFS)



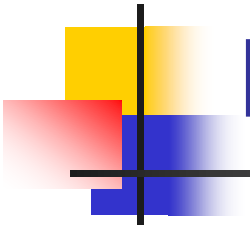
Depth Limited Search (vs. DFS)

- DFS may never terminate as it could follow a path that has no solution on it
- DLS solves this by imposing a depth limit, at which point the search terminates at that particular branch



Depth Limited Search - Observations

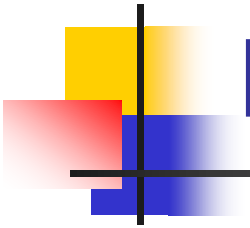
- Can be implemented by the general search algorithm using operators which keep track of the depth
- Choice of depth parameter is important
  - Too deep is wasteful of time and space
  - Too shallow and we may never reach a goal state



Depth Limited Search - Observations

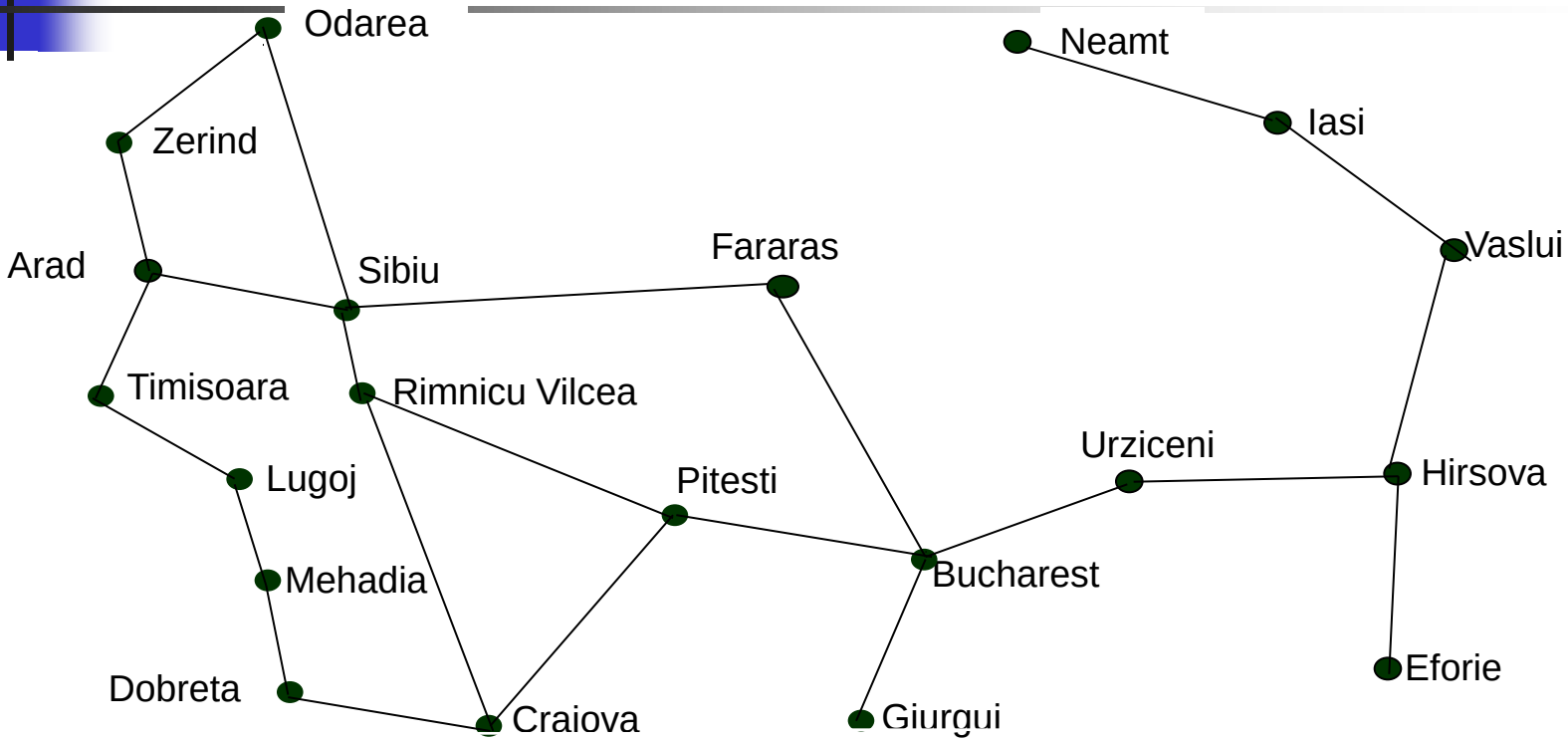
- Completeness
  - If the depth parameter, *l*, is set deep enough then we are guaranteed to find a solution if one exists
    - Therefore it is complete if *l*>=*d* (*d*=depth of solution)

Depth Limited Search - Observations



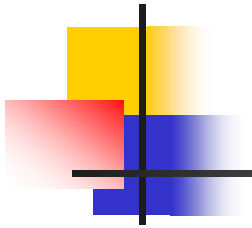
- Space requirements
  - O(*bl*)
- Time requirements
  - O(b^l)
- DLS is not optimal

Map of Romania

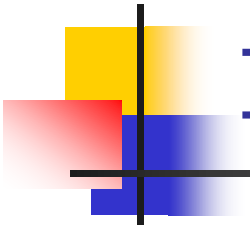


On the Romania map there are 20 towns so any town is reachable in 19 steps

In fact, any town is reachable in 10 steps



Iterative Deepening Search (vs. DLS)



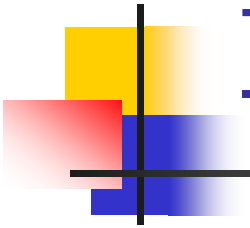
Iterative Deepening Search (vs. DLS)

- The problem with DLS is choosing a depth parameter
- Setting a depth parameter to 19 is obviously wasteful if using DLS
- IDS overcomes this problem by trying depth limits of 0, 1, 2, …, n. In effect it is combining BFS and DFS

Iterative Deepening Search - Observations

- IDS may seem wasteful as it is expanding the same nodes many times
- In fact, IDS expands just 11% more nodes of those by BFS or DLS
  - If $b = 10$, $d = 5$
    - N(IDS) = 123,450, N(BFS) = 100,000
  - Time Complexity = $O(b^d)$
  - Space Complexity = $O(bd)$

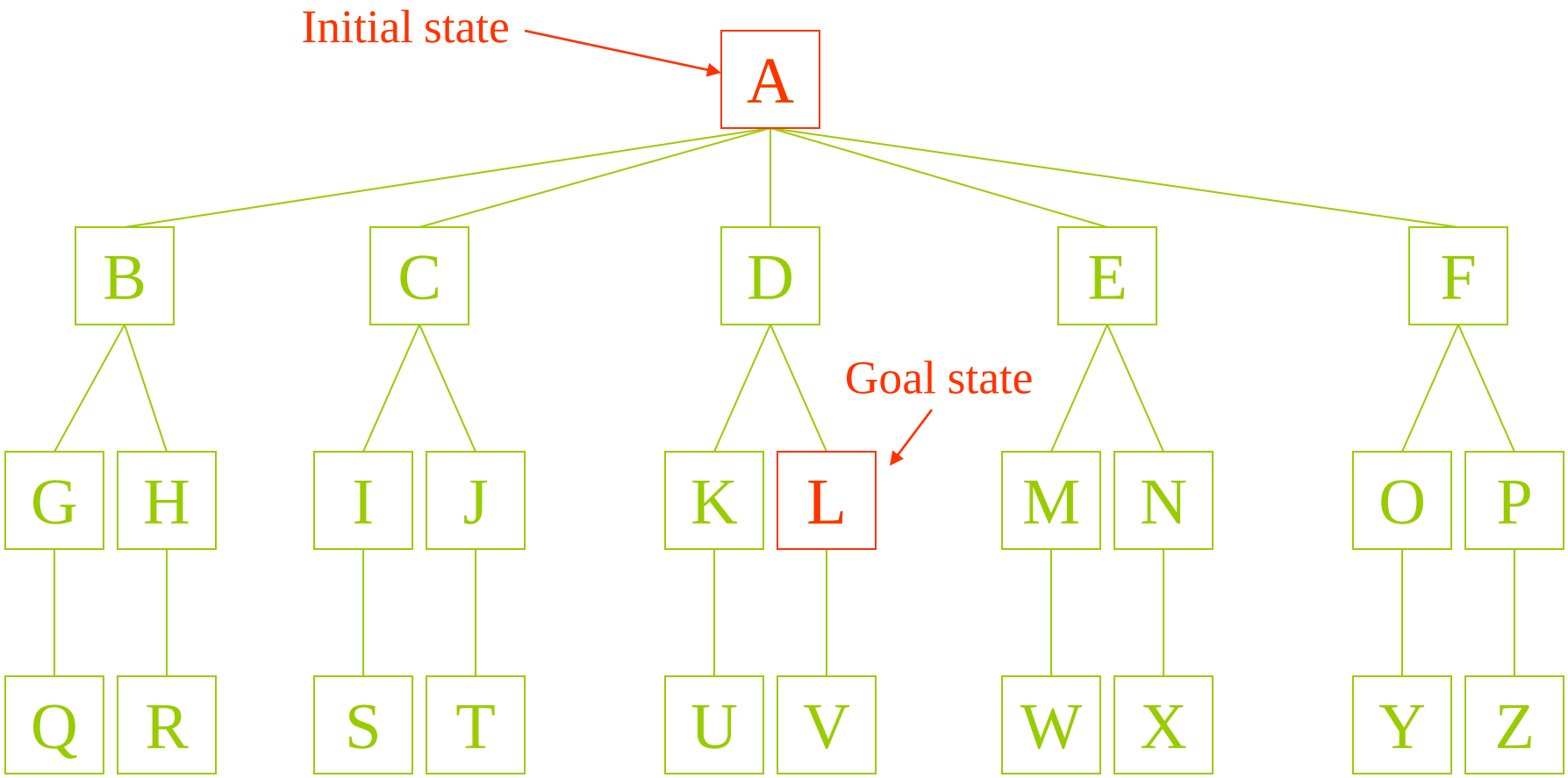
Iterative Deepening Search - Observations



- For large search spaces, where the depth of the solution is not known, IDS is normally the preferred search method

- Self study
  - http://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

The example node set



Press space to see a IDS of the example node set

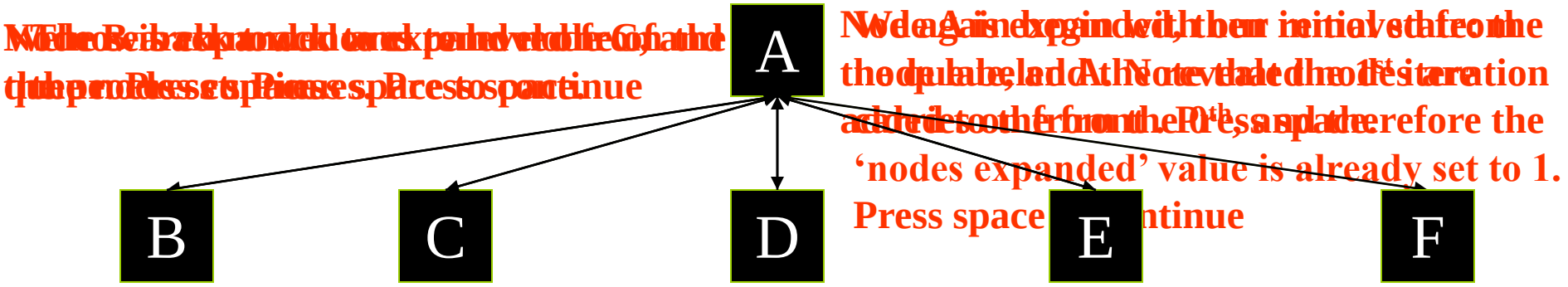
We begin with an initial state node labeled A. This node is placed to the queue. Press space to continue

As this is the 0th iteration of the search, we cannot search past any level greater than zero. This iteration now ends, and we begin the 1st iteration.

Press space to begin the search

Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 1	**Current Action: Expanding**	**Current level: 0**

ITERATIVE DEEPENING SEARCH PATTERN (0th ITERATION)

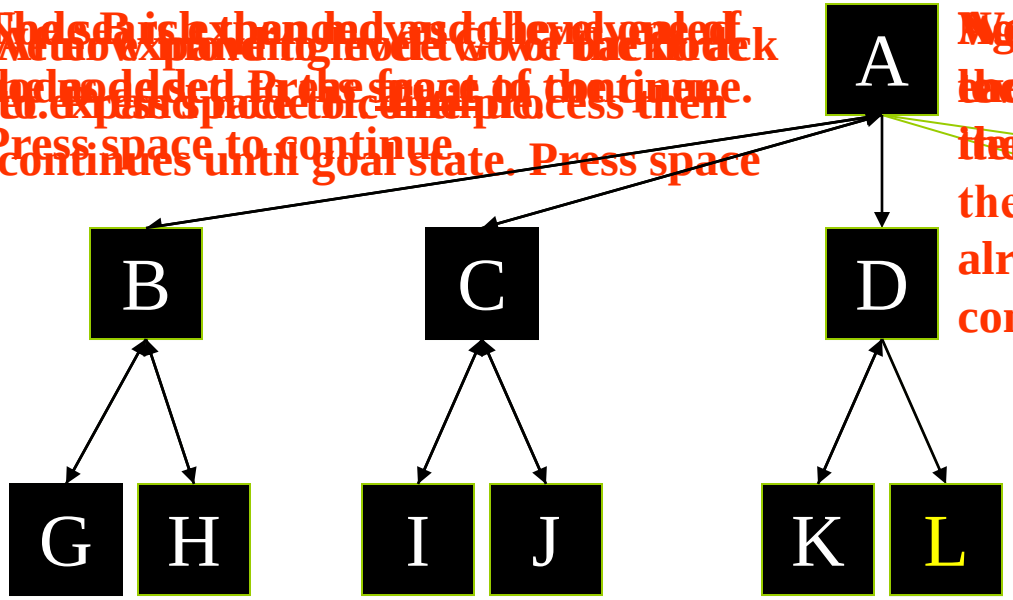


As this is the 1st iteration of the search, we cannot search past any level greater than level one. This iteration now ends, and we begin a 2nd iteration.

Press space to begin the search

Size of Queue: 0	**Queue: Empty**	
Nodes expanded: 7	**Current Action: Expanding**	**Current level: 1**

ITERATIVE DEEPENING SEARCH PATTERN (1st ITERATION)



Node L is located on the second level and the search returns a solution on its second iteration. Press space to end.

Press space to continue the search

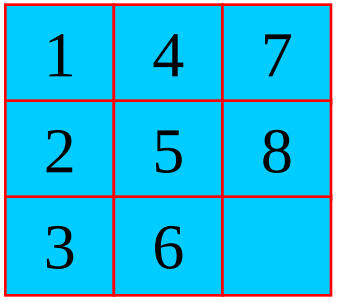
Size of Queue: 0	Queue: Empty	
Nodes expanded: 16	SEARCH FINISHED	Current level: 2

ITERATIVE DEEPENING SEARCH PATTERN (2nd ITERATION)

Repeated States - Three Methods

1. Do not generate a node that is the same as the parent node
   Or
   Do not return to the state you have just come from

2. Do not create paths with cycles in them. To do this we can check each ancestor node and refuse to create a state that is the same as this set of nodes

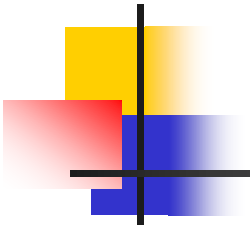


Repeated States - Three Methods

3. Do not generate any state that is the same as any state generated before
   - This requires that every state is kept in memory (meaning a potential space complexity of O(b^d))

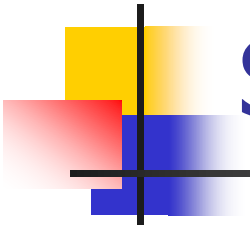
- The three methods are shown in increasing order of computational overhead in order to implement them

1	4	7
2	5	8
3	6	



Evaluation	Breadth First	Uniform Cost	Depth First	Depth Limited	Iterative Deepening
Time	B^D	B^D	B^M	B^L	B^D
Space	B^D	B^D	BM	BL	BD
Optimal?	Yes	Yes	No	No	Yes
Complete?	Yes	Yes	No	Yes, if L >= D	Yes

B = Average branching factor
D = Depth of solution
M = Maximum depth of the search tree
L = Depth Limit



Summary of Blind Search

- Blind searches (Chapter 3 AIMA)
  - Breadth first
  - Uniform cost
  - Depth first
  - Depth limited