Artificial Intelligence

Blind Search

Andrea Torsello

Search

Search plays a key role in many parts of AI. These algorithms provide the conceptual backbone of almost every approach to the systematic exploration of alternatives.

One general approach to problem solving in AI is to reduce the problem to be solved to one of searching a graph.

To use this approach, we must specify what are the states, the actions and the goal test.

Our goal is to plan a series of action that takes us from an initial state to a goal state.

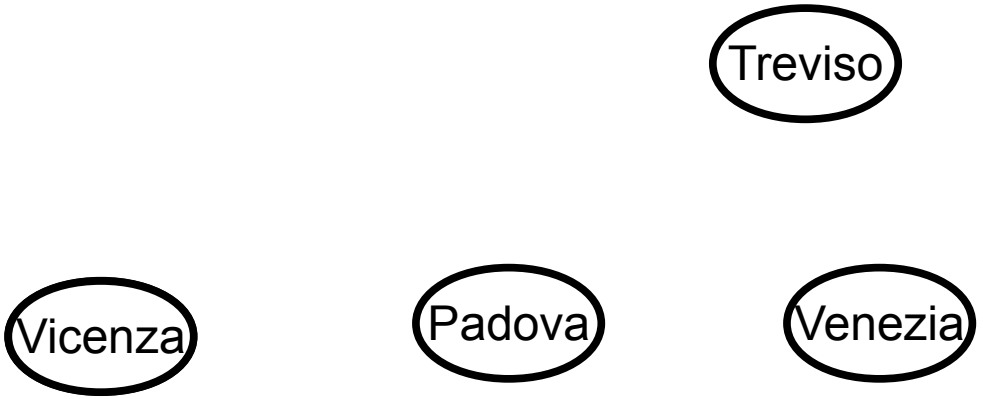
The search can be

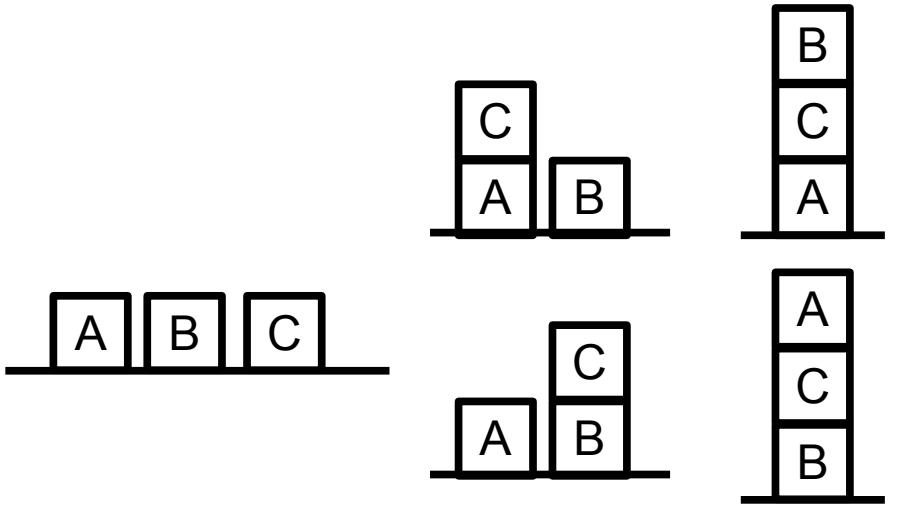
- **Uninformed (or blind)**: I know nothing more than the neighbors of the current state (states reachable through a single action)
- **Informed (or heuristic)**: I have a means to assess progress toward the goal

Graph-Based Search

What are the **states**? (all relevant aspects of the problem)

- Arrangement of parts (to plan an assembly)
- Position of Trucks (to plan package distribution)
- Cities (to plan a trip)
- Set of facts (e.g. to prove theorems)

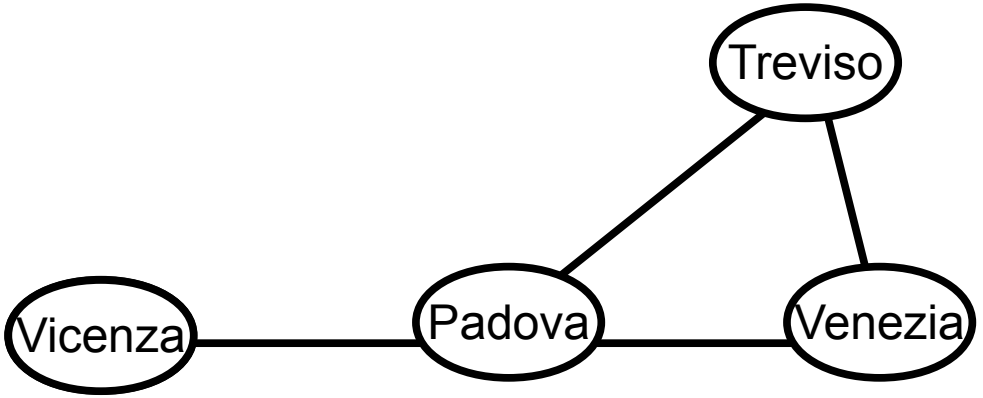


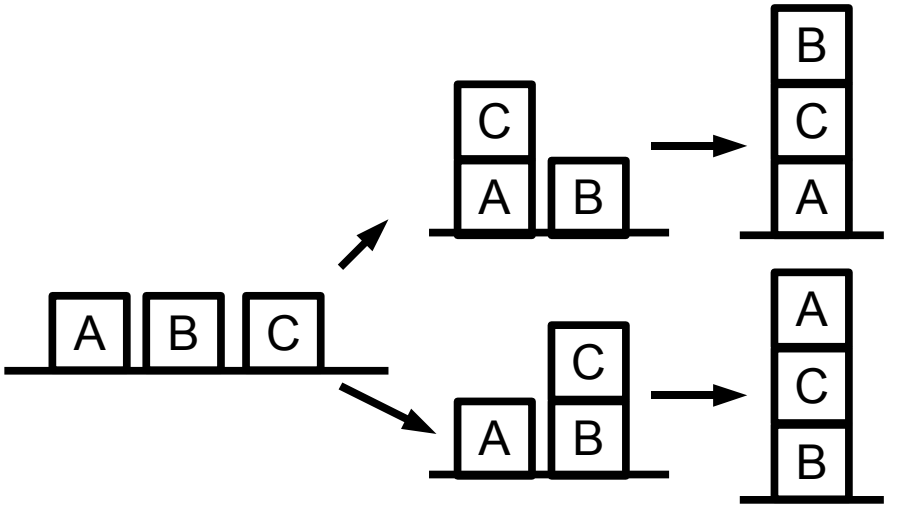


Graph-Based Search

What are the **actions** (operations)? (neighboring states)

- Assemble two parts
- Move a truck to a new position
- Drive to a new city
- Apply a theorem to derive new fact

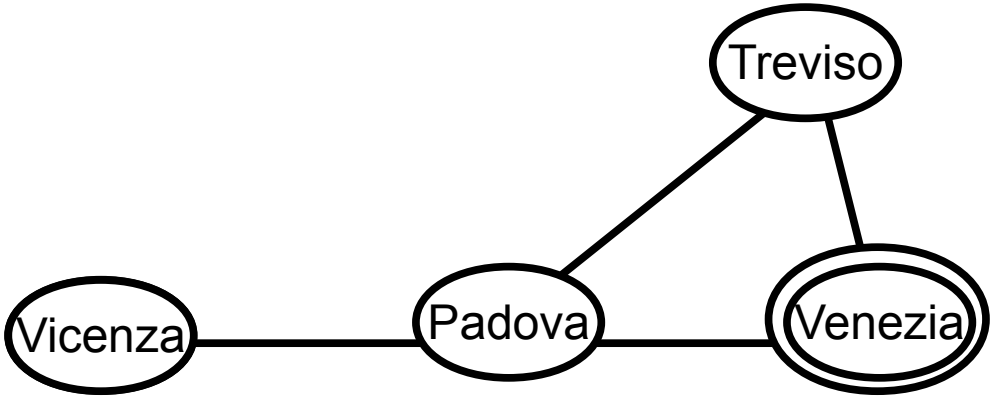


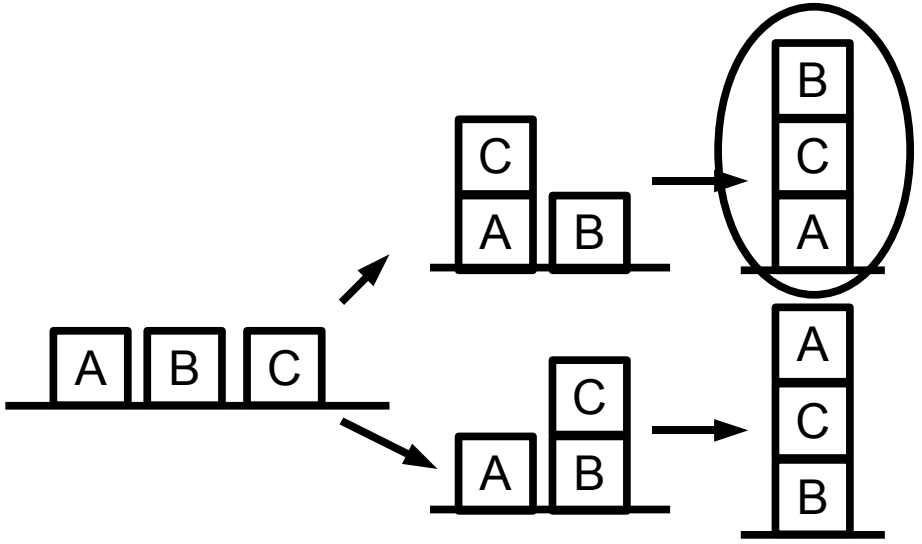


Graph-Based Search

What is the **goal test**? (Condition for success)

- All parts in place
- All package delivered
- Destination reached
- Derived goal fact

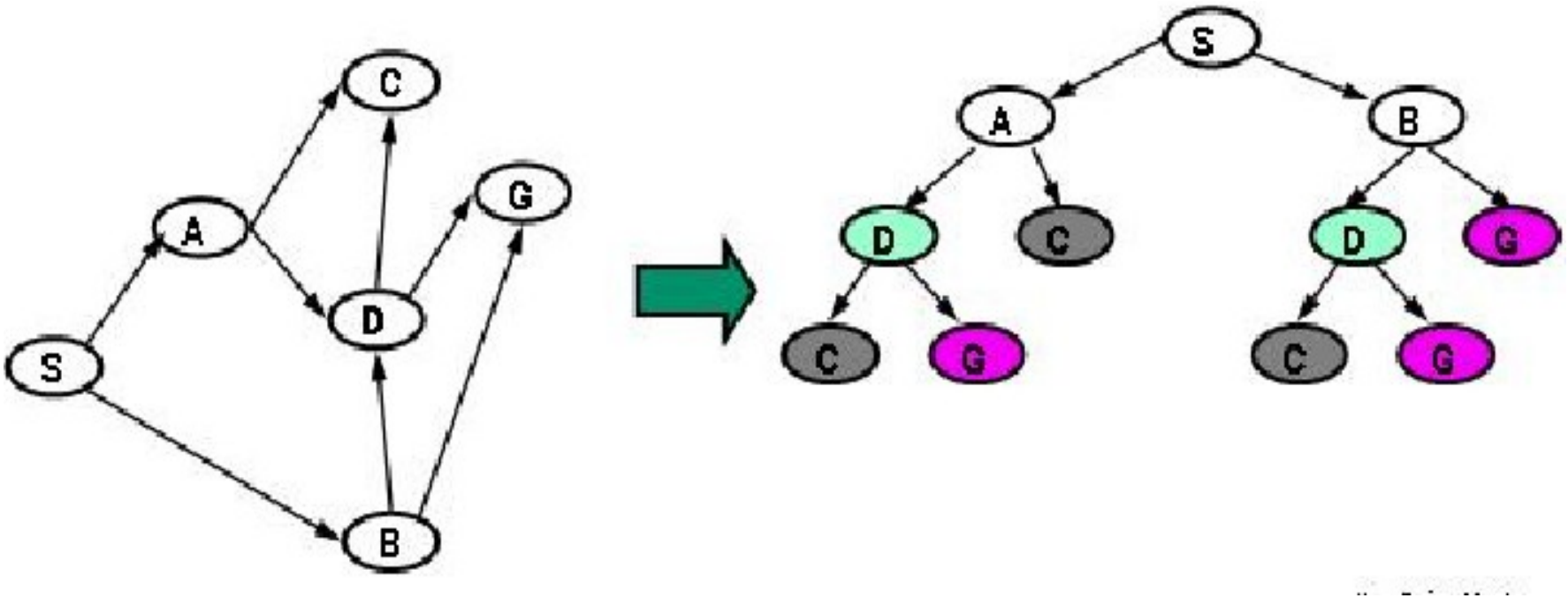




Graph Search as Tree Search

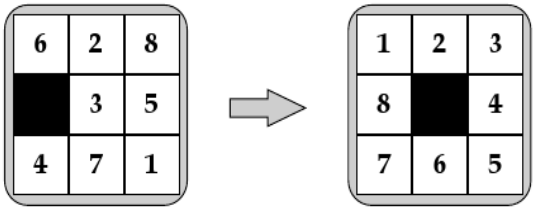
We can turn graph search problems into tree search problems by

- Replacing undirected links by 2 directed links (back and forward)
- Avoiding loops in path (or keeping track of visited nodes)

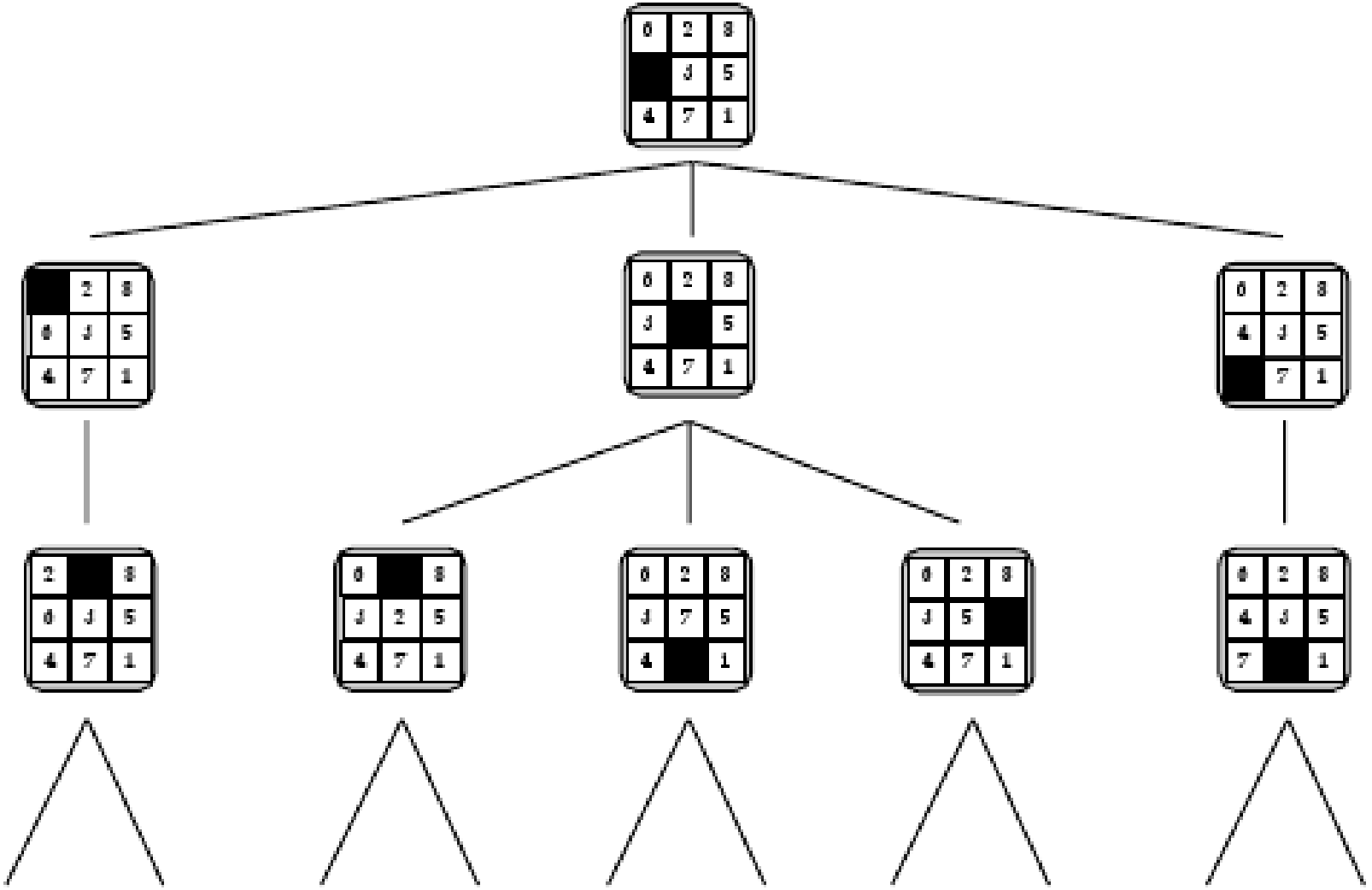


Example: The Game of 8

Goal: move the squares from initial to final configuration



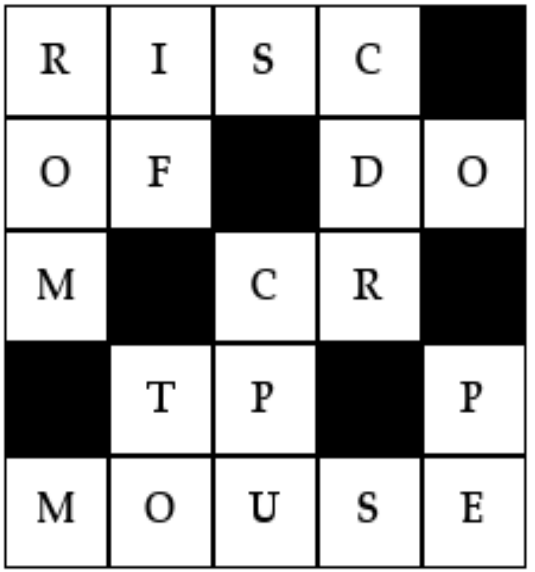
The search tree



Example: Crosswords

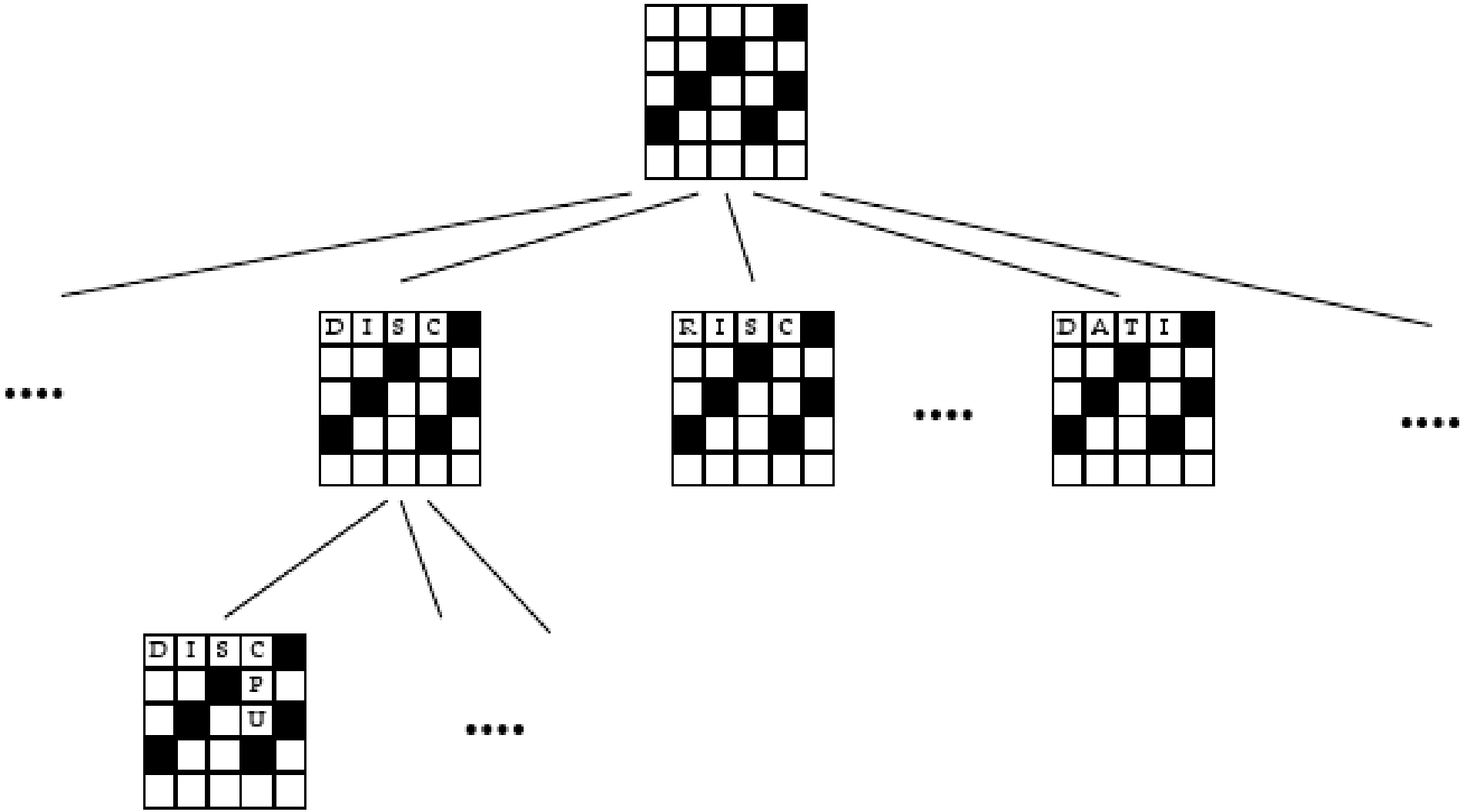
There are 26^{19} possible ways to fill the boxes of the crosswords puzzle on the right, but only a tiny fraction is legal,

i.e., forms real words

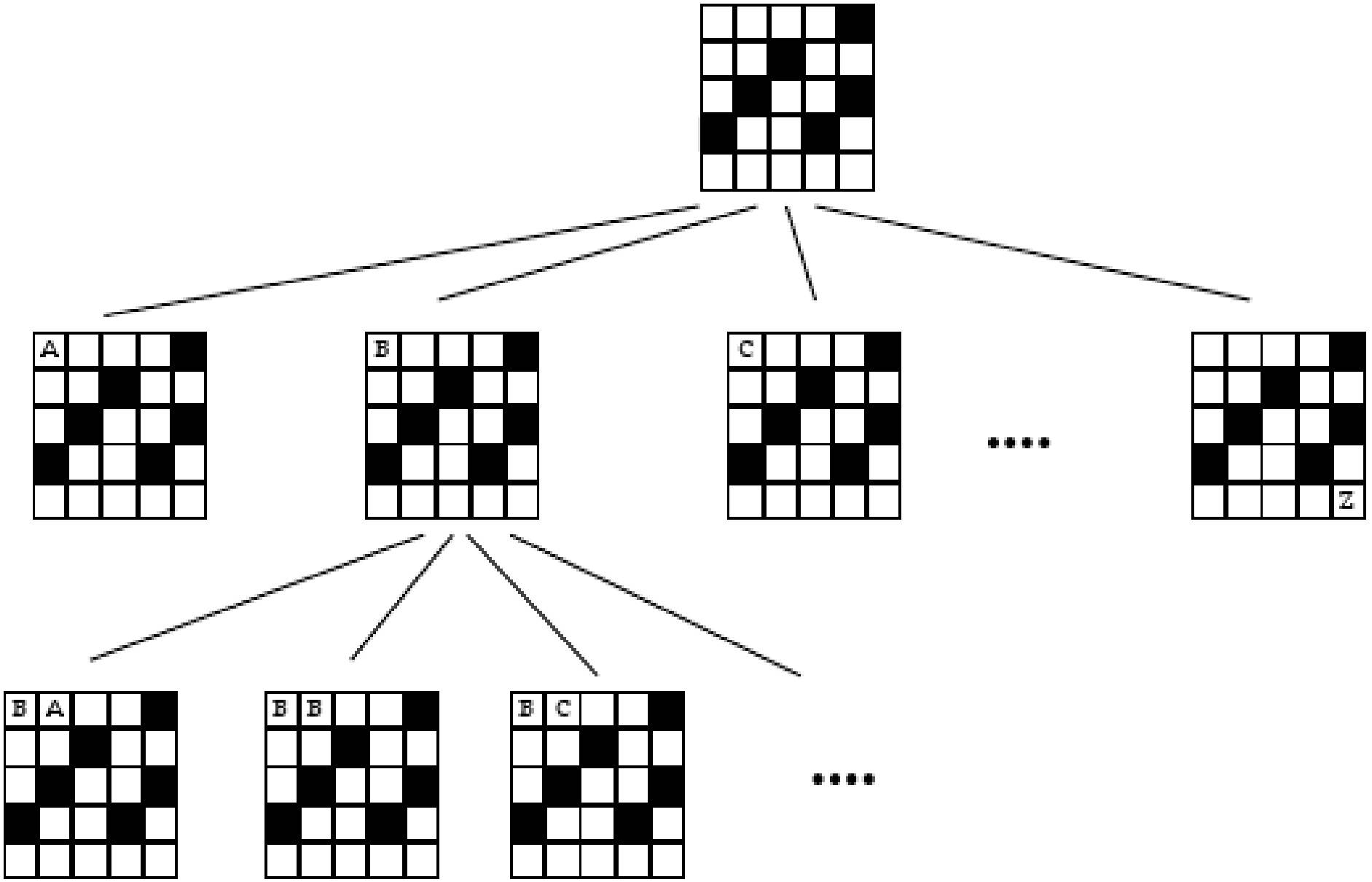


There are several ways in which you can construct the search space, depending on your definition of states and actions

Crosswords: State Space 1



Crosswords: State Space 2



Evaluation

There are 4 criteria to evaluate a search algorithm:

Completeness:

- Will the algorithm find a solution if one exists?

Optimality:

- Will the algorithm find the “best” solution when more than one exist?

Temporal Complexity:

- How long will it take to find a solution?

Spatial Complexity:

- How much memory will it require?

Stab of Search Procedure

Let ***L*** be a list of **visited** but not **expanded** nodes

1) Initialize ***L*** with the initial state
2) If ***L*** is empty, **FAIL**, else *extract* a node ***n*** from ***L***
3) If ***n*** is a goal node, **SUCCEED** and return the path from the initial state to ***n***
4) Remove ***n*** from ***L*** and *insert* all the children of ***n***
5) Goto 2)

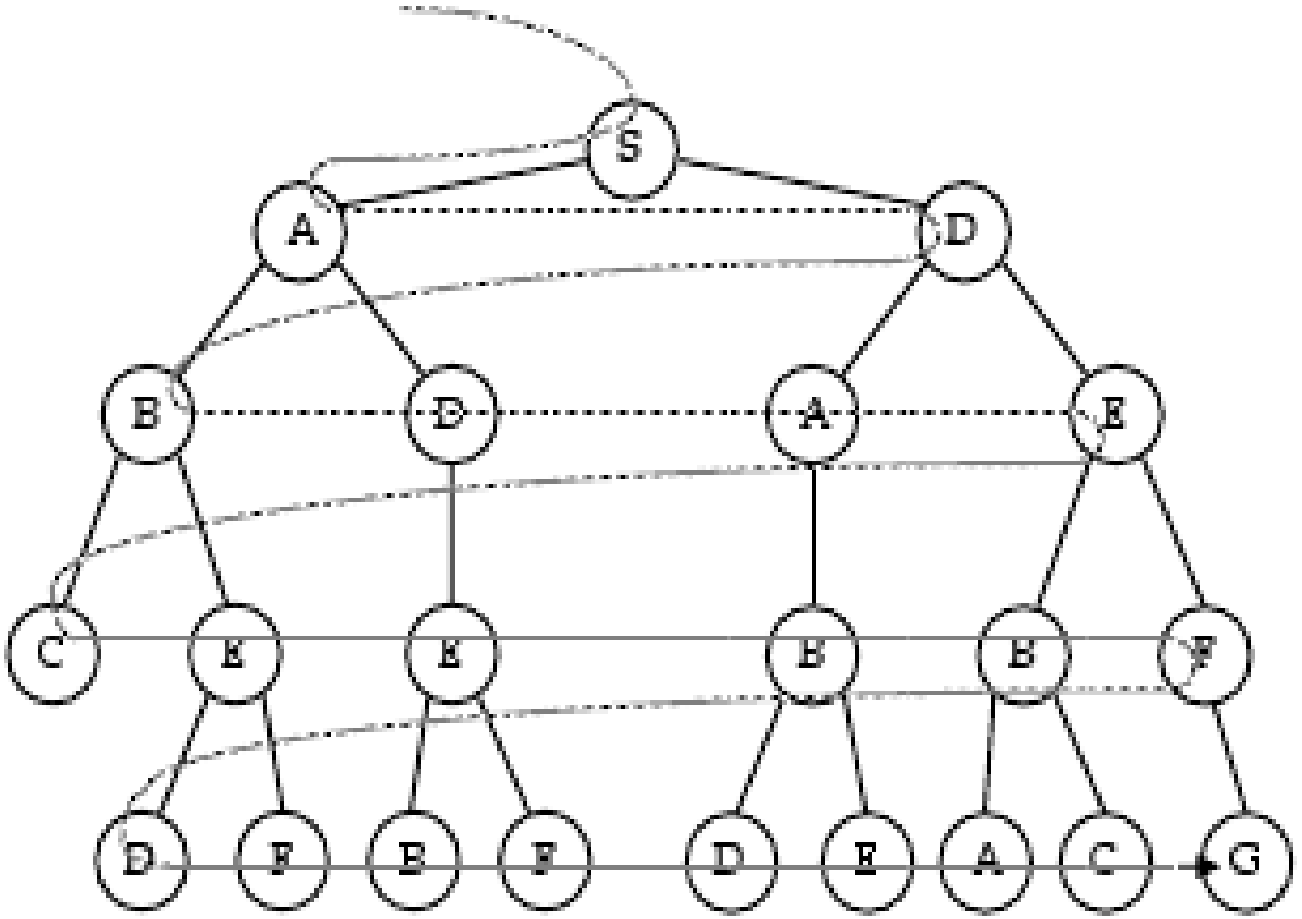
The type of search depend on how you extract and insert nodes from and into ***L***

Breadth-First Search

Search is performed by extracting from the front and inserting on the back of the list (FIFO)

The algorithm proceed level by level, hence it is

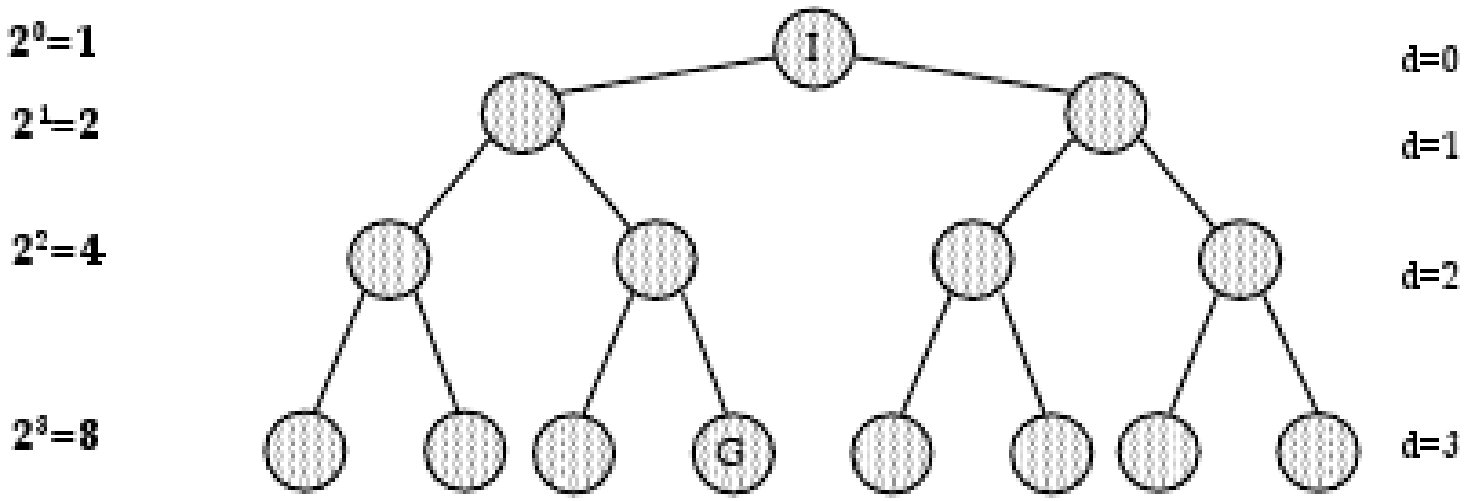
- Complete
- Optimal (will find the solution at smallest depth)



Breadth-First: Temporal Complexity

Simplifying hypotheses:

- The search tree has constant branching factor *b*
- The first goal is at depth *d*



Observation: the number of nodes at level *d* is b^d

Breadth-First: Temporal Complexity

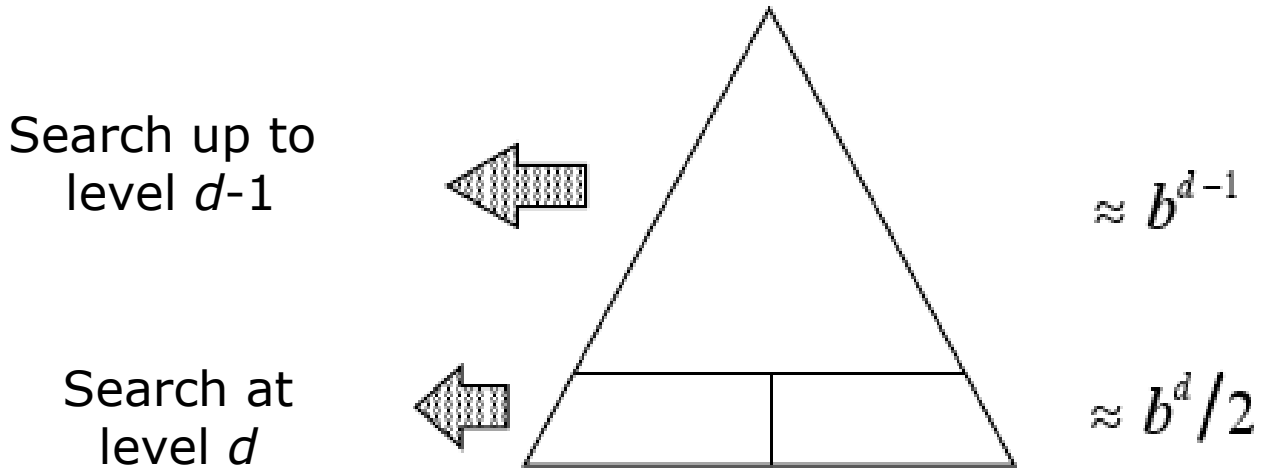
The number of nodes to examine to reach depth *d* is:

$$1 + b + b^2 + \ldots + b^{d-1} = \frac{b^d - 1}{b - 1}$$

The average number of nodes to examine at level *d* is:

$$\frac{1 + b^d}{2}$$

Adding the two terms we obtain: $\frac{b^{d+1} + b^d + b - 3}{2(b-1)} = O(b^d)$



Note: time complexity is dominated by search at the last level

Breadth-First: Temporal Complexity

Before the algorithms examines the first node at depth k, it must have memorized all of them (b^k)

The memory required to reach the goal node at level d is $O(b^d)$

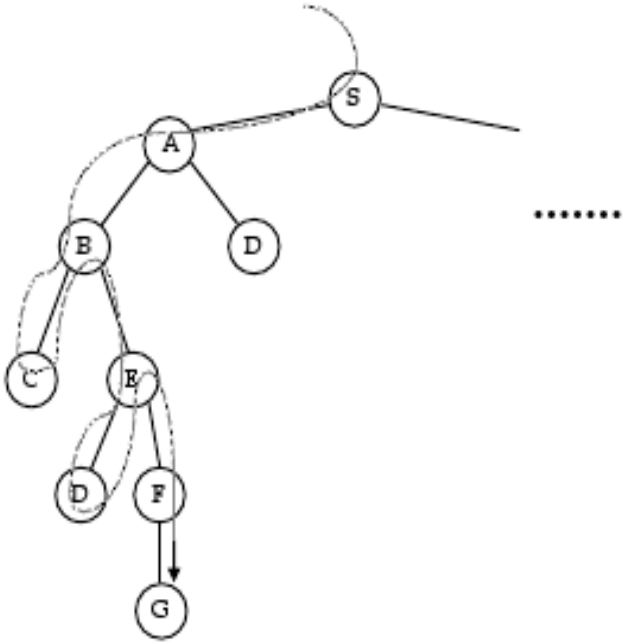
Example

Suppose to perform a search on a tree with branching factor b=10. The program computes 1000 nodes per second and requires 100 byte per node

Level	Nodes	Time	Memory
2	111	0.1 sec	11 Kb
6	10^6	18 min	111 Mb
10	10^{10}	128 giorni	11 Gb
14	10^{14}	3500 anni	11.111 Tb

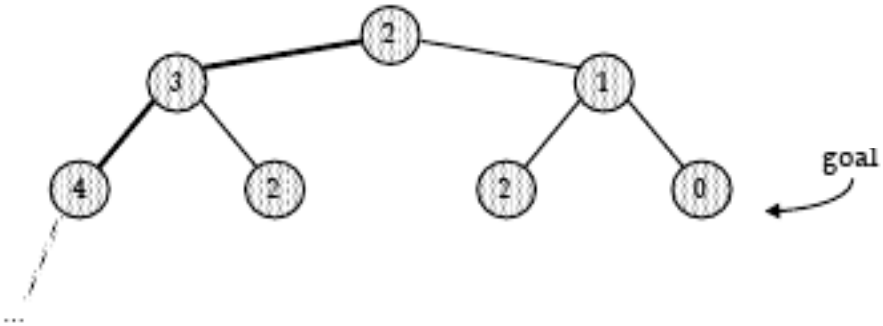
Depth-First Search

Search is performed by extracting and inserting from the front of the list (LIFO)



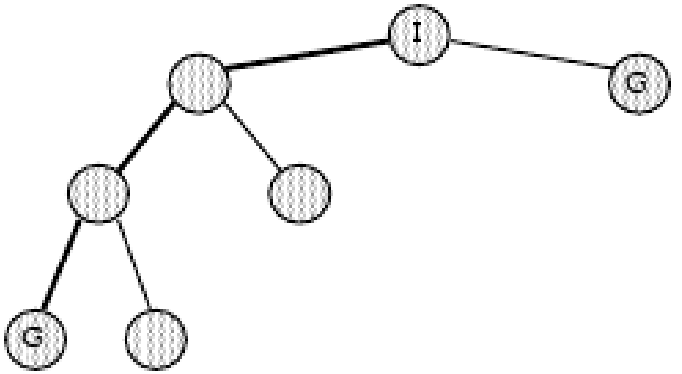
If the tree has infinite depth, the algorithm might get stuck in an infinite branch and not find a solution even when one exists

- The algorithm is not complete



If the problem has more than one solution the algorithm is not guaranteed to find the one at minimum depth

- The algorithm is not optimal



Depth-First Search: Temporal Complexity

Simplifying hypotheses:

- The search tree has constant branching factor *b*
- The tree has depth *d*
- There is only one goal and it is at depth *d*

In the best case the goal is at the far left of the tree. Hence, the numeber of nodes to examine is:

$$d+1$$

In the worst case it is at the far right:

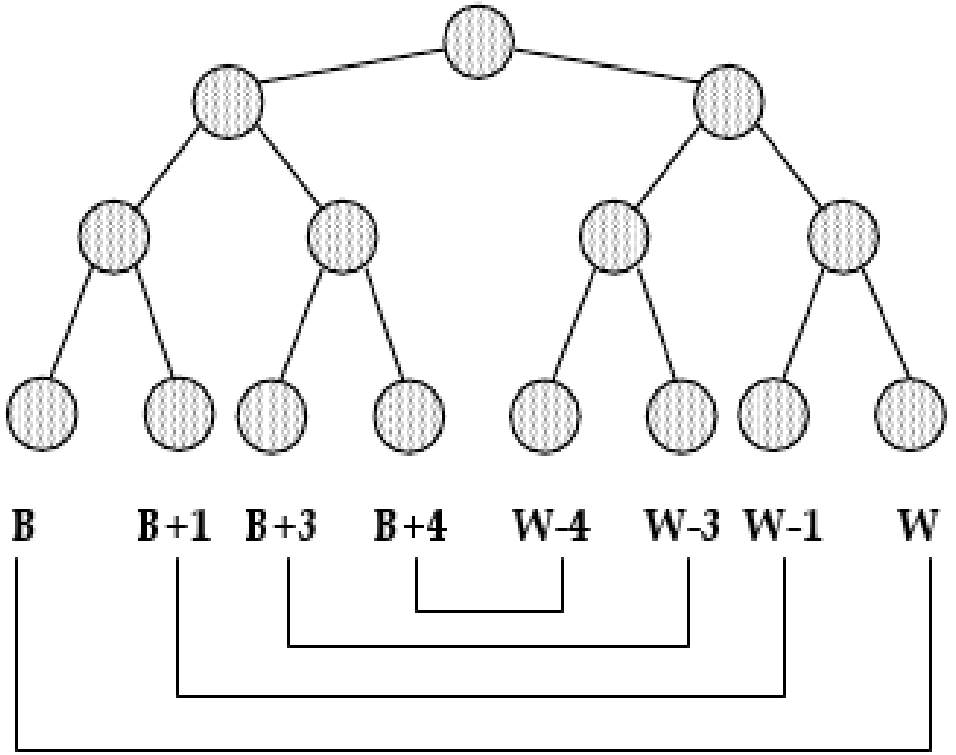
$$1+\ldots+b^d = \frac{b^{d+1}-1}{b-1}$$

Taking the average we have:

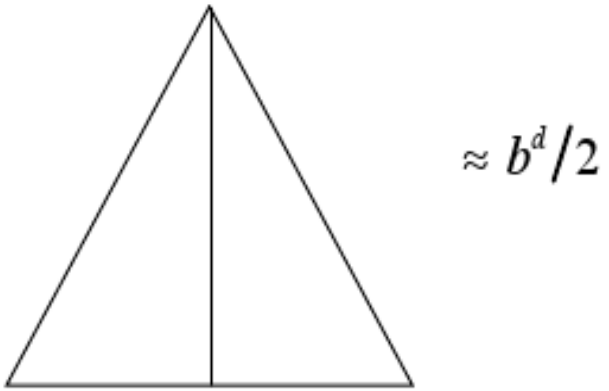
$$\frac{b^{d+1}+bd+b-d-2}{2(b-1)}$$

Depth-First Search: Temporal Complexity

Is it reasonable to take the average?



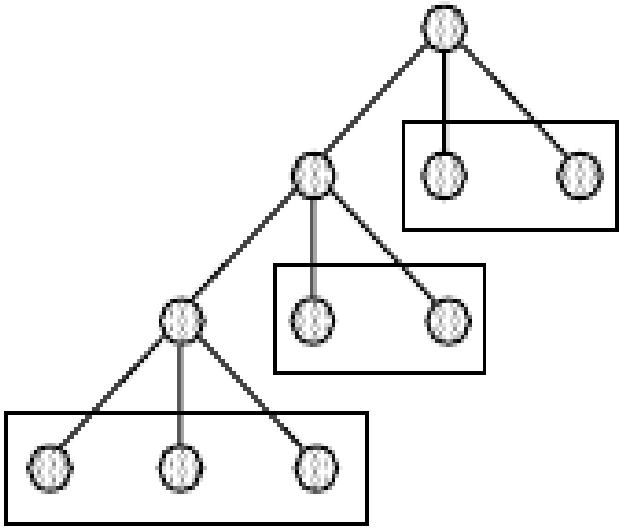
For large d, the average search time is around $b^d/2=O(b^d)$, i.e., the time to visit half the tree



Depth-First Search: Spatial Complexity

The maximum memory required is

$$d(b-1)+1 = O(db)$$



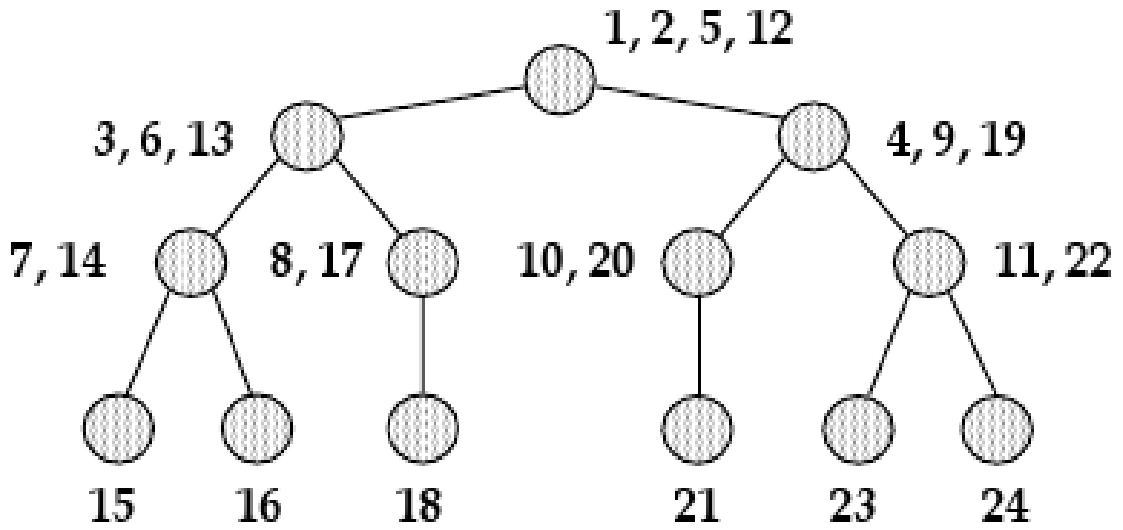
Iterative deepening

Basic idea:

- perform depth-first search up-to depth ***C***
- Increase maximum depth ***C*** until goal is found

Let L be the list of visited but not expanded node, and ***C*** the maximum depth

1) Let ***C***=0
2) Initialize ***L*** to the initial state (only)
3) If ***L*** is empty increase ***C*** and goto 2),
   else *extract* a node ***n*** from the *front* of L
4) If ***n*** is a goal node, **SUCCEED** and return the path from the initial state to ***n***
5) Remove ***n*** from ***L***. If the level is smaller than ***C***, *insert* all the children of ***n*** at the *fr*
6) Goto

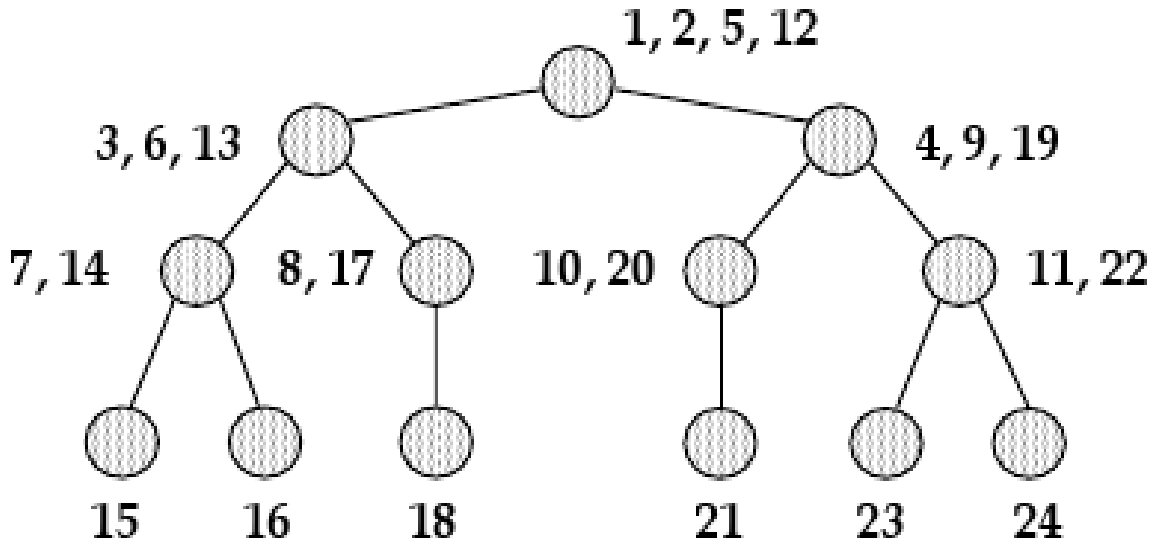


NB: the numbers represent the expansion order of the nodes

Iterative Deepening

The algorithm proceeds level by level, thus, just like breadth first, it is *complete* and *optimal*

Example:



Here the numbers represent the expansion order of the nodes

Iterative Deepening: Temporal Complexity

In the last iteration (at level d), the average nodes to be examined is:

$$\frac{b^{d+1} + bd + b - d - 2}{2(b-1)}$$

In each previous iteration the number of nodes the algorithm examined is:

$$\frac{b^{C+1} - 1}{b - 1}$$

Hence, in total, from all the previous iterations, we have:

$$\sum_{j=0}^{d-1} \frac{b^{j+1} - 1}{b-1} = \frac{b^{d+1} - bd - b + d}{(b-1)^2}$$

Adding we have

$$\frac{b^{d+2} + b^{d+1} + b^2 d + b^2 - 4bd - 5b + 3d + 2}{2(b-1)^2} = O(b^d)$$

Iterative Deepening: Spatial Complexity

Since at each iteration the algorithm performs a depth-first search, the memory requirement is the same as Depth-First:

$$d(b-1)+1 = O(db)$$

	Breadth-First	Depth-First	Iterative Deepening
Complete	yes	no	yes
Optimal	yes	no	yes
Time	b^d	b^d	b^d
Space	b^d	bd	bd