Artificial Intelligence

Blind Searches

Is ant search for food is Blind Searches??

Blind Searches - Characteristics

- Simply searches the State Space
- Can only distinguish between a goal state and a non-goal state
- Sometimes called an uninformed search as it has no knowledge about its domain

Blind Searches - Characteristics

- Blind Searches have no preference as to which state (node) that is expanded next
- The different types of blind searches are characterised by the order in which they expand the nodes.
- This can have a dramatic effect on how well the search performs when measured against the four evaluation criteria.

Blind Searches - Why Use

- We may not have any domain knowledge we can give the search
- We may not want to implement a specific search for a given problem. We may prefer just to use a blind search

Example: the 8-puzzle.

- Given: a board situation for the 8-puzzle:

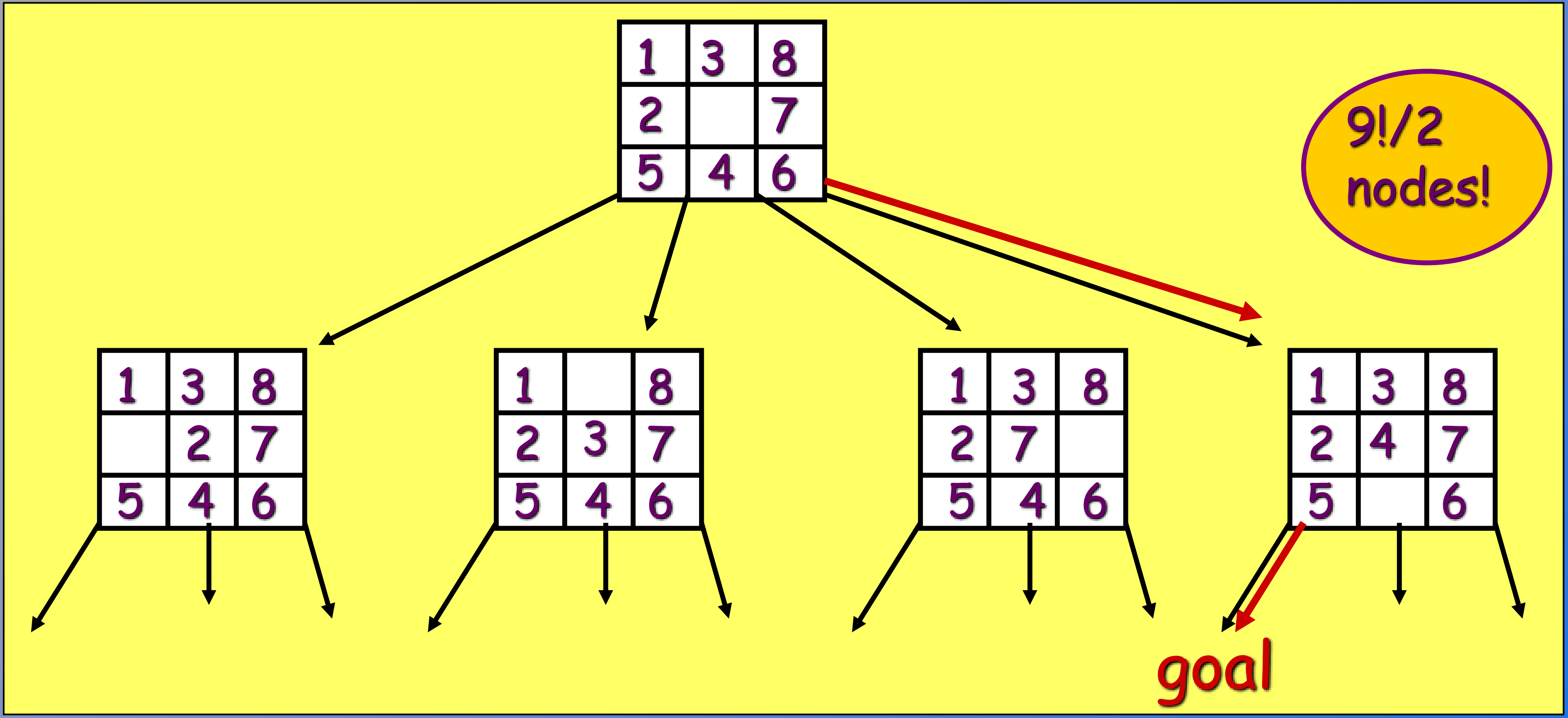
1	3	8
2		7
5	4	6

- Problem: find a sequence of moves (allowed under the rules of the 8-puzzle game) that transform this board situation in a desired goal situation:

1	2	3
8		4
7	6	5

The (implicit) search tree

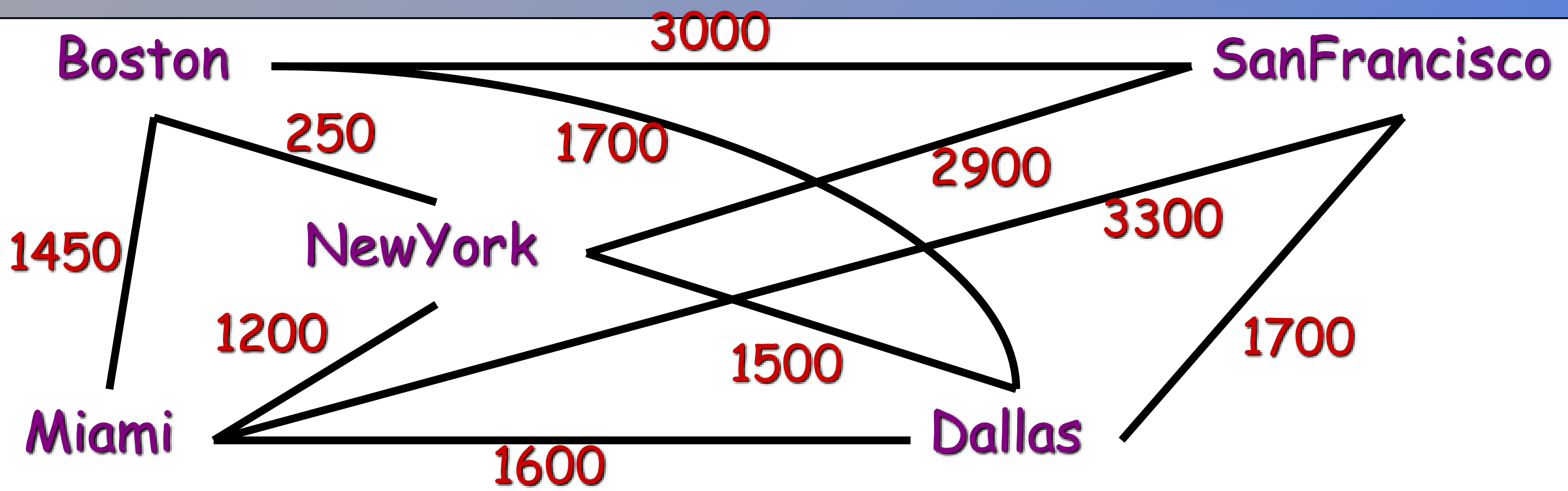
- Each state-space representation defines a search tree:



- But this tree is only IMPLICITLY available !!

Any path, versus shortest path, versus best path:

Ex.: Traveling salesperson problem:

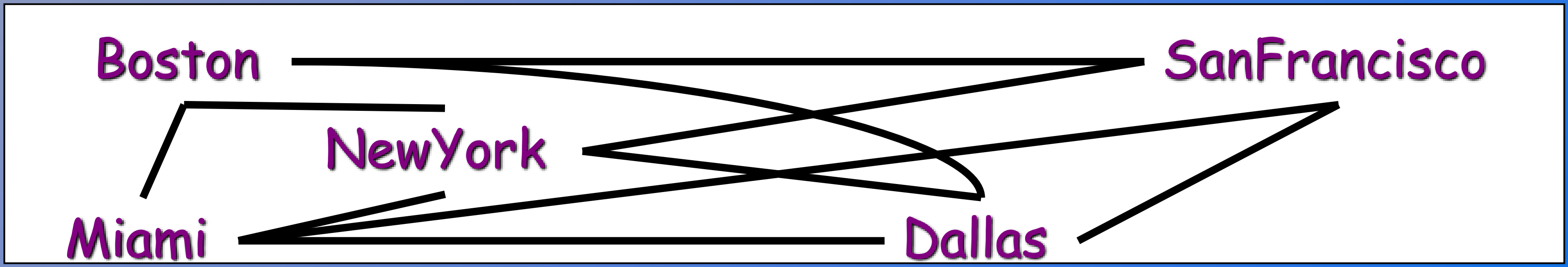


◎ Find a sequence of cities ABCDEA such that the total distance is MINIMAL.

Best path problem

State space representation:

- **State:**
  - ➔ the list of cities that are already visited
    - ◆ Ex.: (NewYork, Boston)
- **Initial state:**
    - ◆ Ex.: (NewYork)
- **Rules:**
  - ➔ add 1 city to the list that is not yet a member
  - ➔ add the first city if you already have 5 members
- **Goal criterion:**
  - ➔ first and last city are equal



BLIND Search Methods

Methods that do not use any specific knowledge about the problem:

Depth-first

Breadth-first

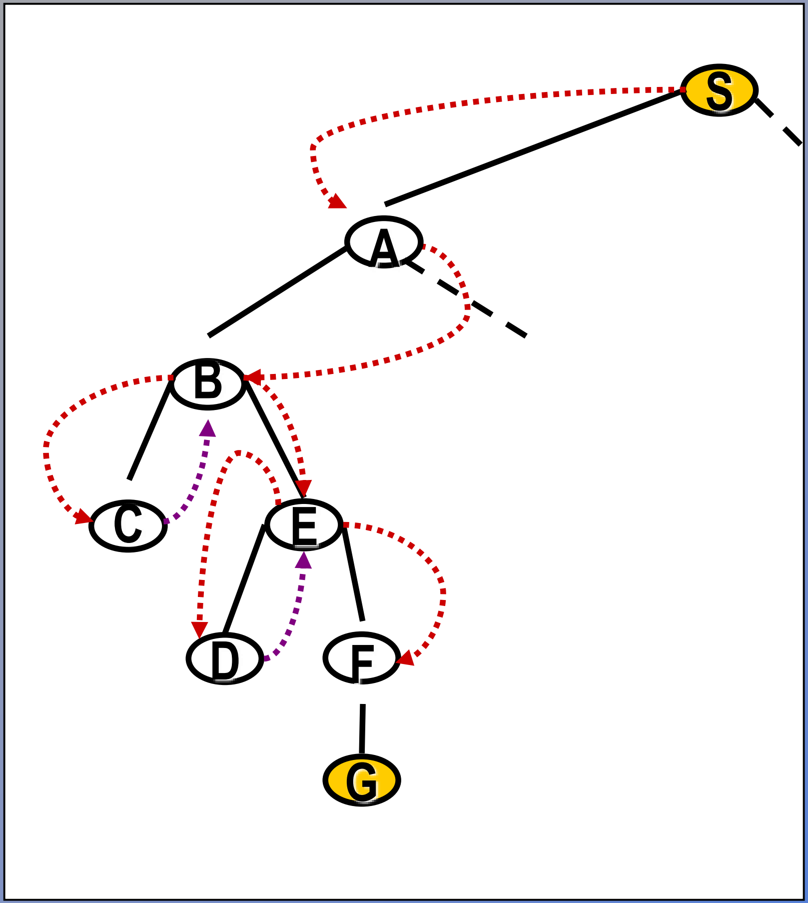
Non-deterministic search

Iterative deepening

Depth-first search

Expand the tree as deep as possible,
returning to upper levels when needed.

Depth-first search = Chronological backtracking



- Select a child
  - ➔convention: left-to-right
- Repeatedly go to next child, as long as possible.
- Return to left-over alternatives (higher-up) only when needed.

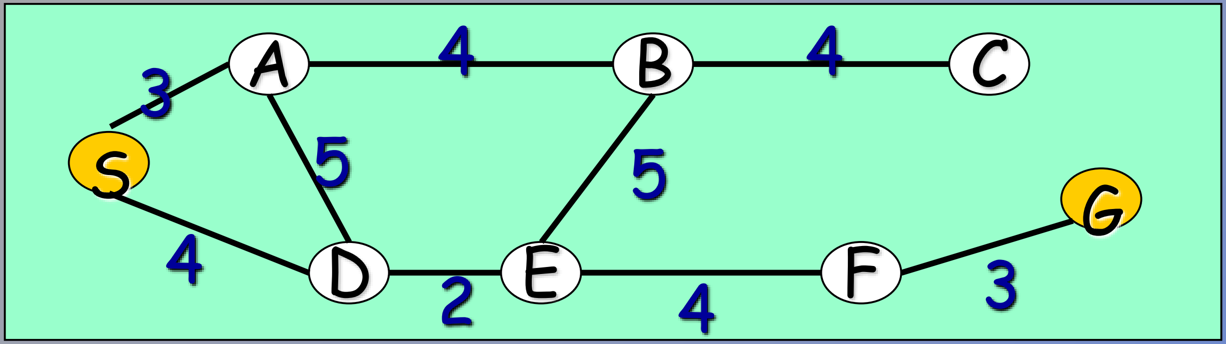
Depth-first algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO
   - remove the first path from the QUEUE;
   - create new paths (to all children);
   - reject the new paths with loops;
   - add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;



1. QUEUE <-- path only containing the root;

2. WHILE { QUEUE is not empty
AND goal is not reached

DO { remove the first path from the QUEUE;
create new paths (to all children);
reject the new paths with loops;
add the new paths to front of QUEUE;

3. IF goal reached
THEN success;
ELSE failure;

Trace of depth-first for running example:

(S)	S removed, (SA,SD) computed and added
(SA, SD)	SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
(SAB,SAD,SD)	SAB removed, (SABA,SABC,SABE) computed, (SABC,SABE) added
(SABC,SABE,SAD,SD)	SABC removed, (SABCB) computed, nothing added
(SABE,SAD,SD)	SABE removed, (SABEB,SABED,SABEF) computed, (SABED,SABEF)added
(SABED,SABEF,SAD,SD)	SABED removed, (SABEDS,SABEDA.SABEDE) computed, nothing added
(SABEF,SAD,SD)	SABEF removed, (SABEFE,SABEFG) computed, (SABEFG) added
(SABEFG,SAD,SD)	goal is reached: reports success

Evaluation criteria:

- Completeness:
  - ➔ Does the algorithm always find a path?
    - (for every NET such that a path exits)
- Speed (worst time complexity) :
  - ➔ What is the highest number of nodes that may need to be created?
- Memory (worst space complexity) :
  - ➔ What is the largest amount of nodes that may need to be stored?
- Expressed in terms of:
    - d = depth of the tree
    - b = (average) branching factor of the tree
    - m = depth of the shallowest solution

<u>Note: approximations !!</u>

- In our complexity analysis, we do not take the built-in <u>loop-detection</u> into account.

- The results only 'formally' apply to the variants of our algorithms WITHOUT loop-checks.
- Studying the effect of the loop-checking on the complexity is hard:
  - ➔ overhead of the checking MAY or MAY NOT be compensated by the reduction of the size of the tree.

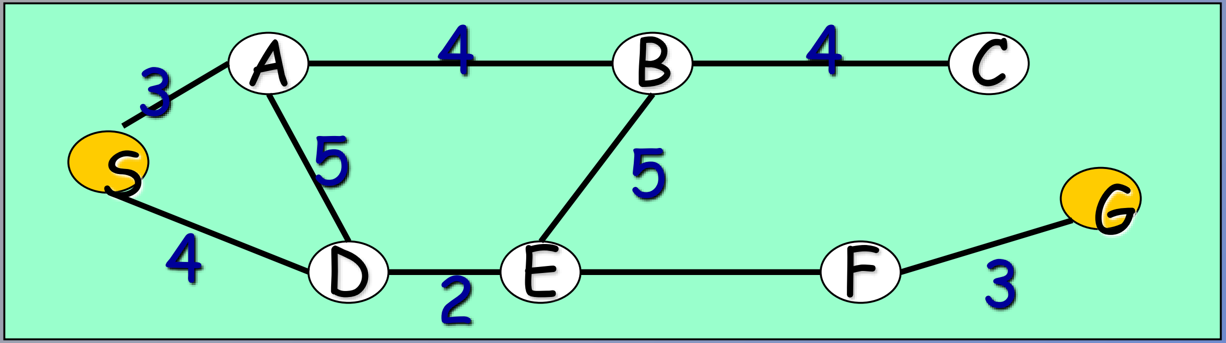
- <u>Also</u>: our analysis DOES NOT take the length (space) of representing paths into account !!

Completeness (depth-first)

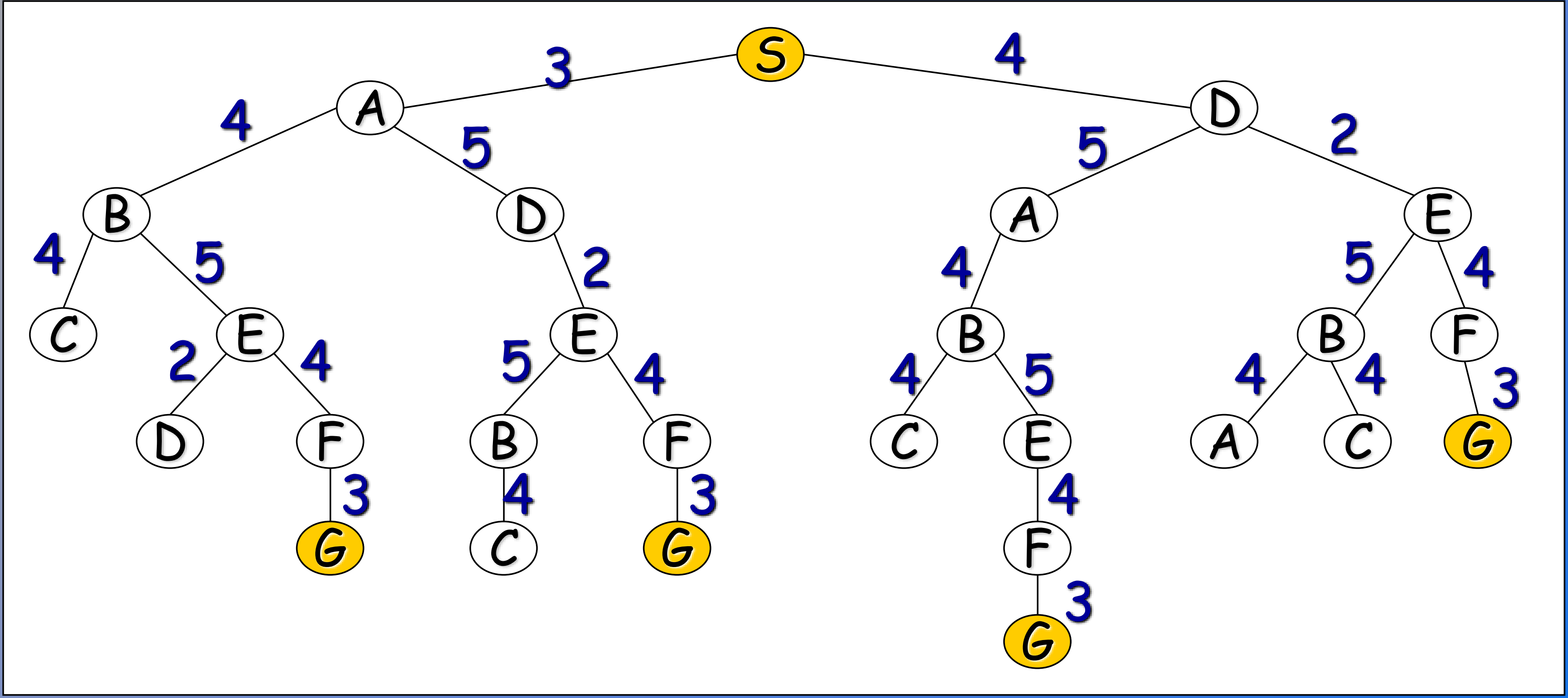
- Complete for FINITE (implicit) NETS.
  - ➔ (= NETS with finitely many nodes)

- IMPORTANT:
  - ➔ This is due to integration of LOOP-checking in this version of Depth-First (and in all other algorithms that will follow) !
    - IF we do not remove paths with loops, then Depth-First is not complete (may get trapped in loops of a finite NET)

- Note: does NOT find the shortest path.



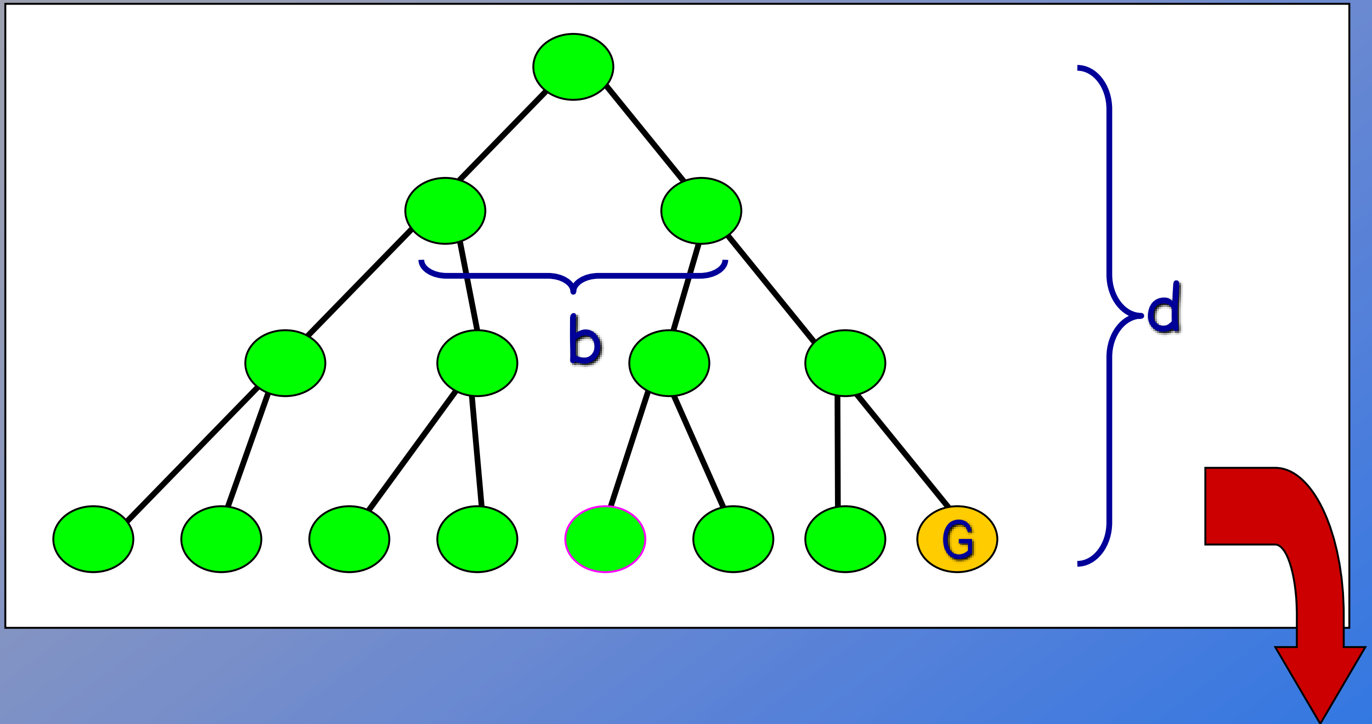
S
A
B
C
D
E
F
G
3
4
4
5
5
4
2
3



S
A
D
B
C
E
F
G
3
4
5
2

Speed (depth-first)

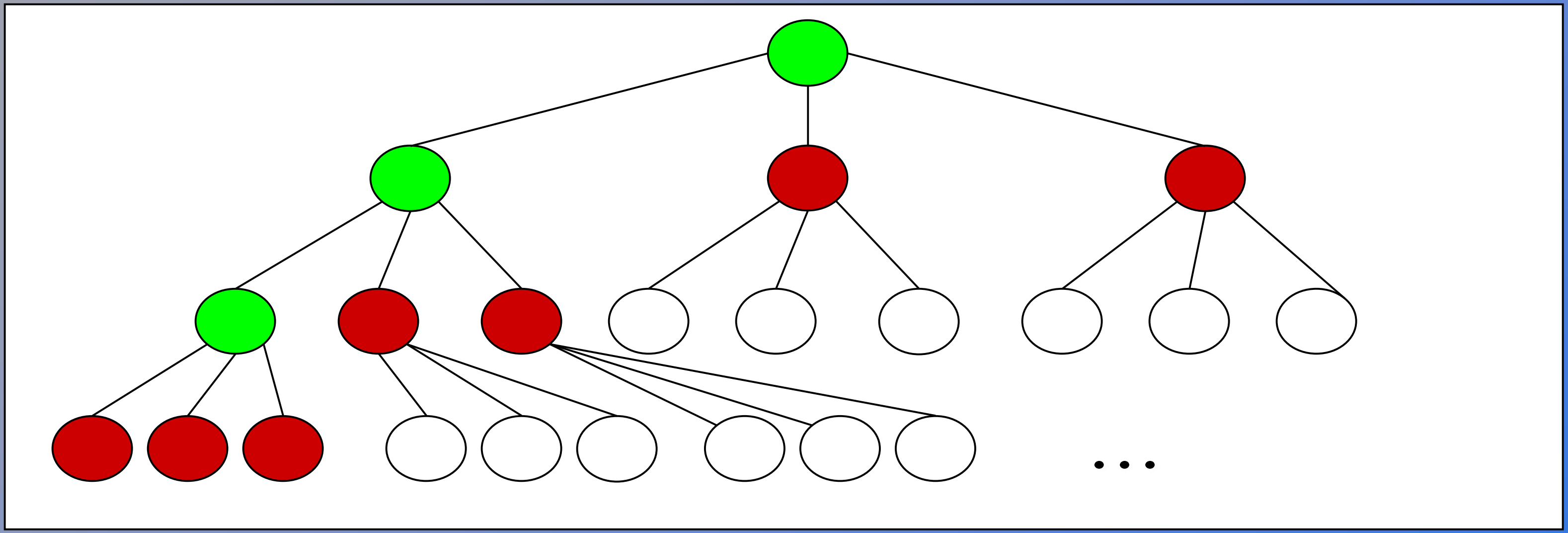
- In the worst case:
  - → the (only) goal node may be on the right-most branch,

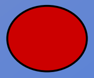


- Time complexity $== b^d + b^{d-1} + \ldots + 1 = \dfrac{b^{d+1} - 1}{b - 1}$
- Thus: $O(b^d)$

Memory (depth-first)

- Largest number of nodes in QUEUE is reached in bottom left-most node.
- Example: d = 3, b = 3 :

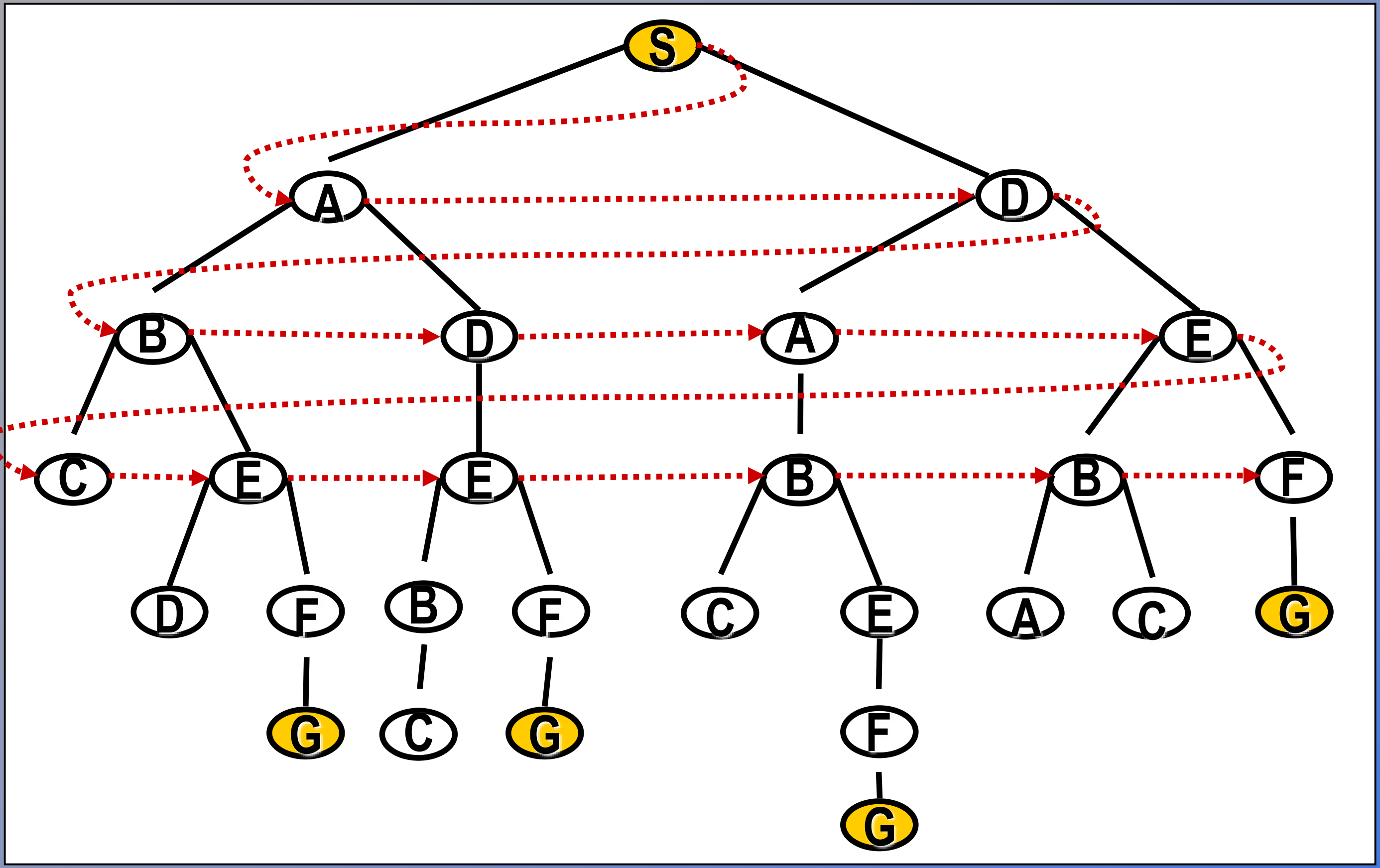


- QUEUE contains all  nodes. Thus: 7.
- In General: ((b-1) * d) + 1
- Order: O(d*b)

Breadth-first search

Expand the tree layer by layer, progressing in depth.

Breadth-first search:



- Move downwards, level by level, until goal is reached.

Breadth-first algorithm:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to back of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

ONLY DIFFERENCE !

Trace of breadth-first for running example:

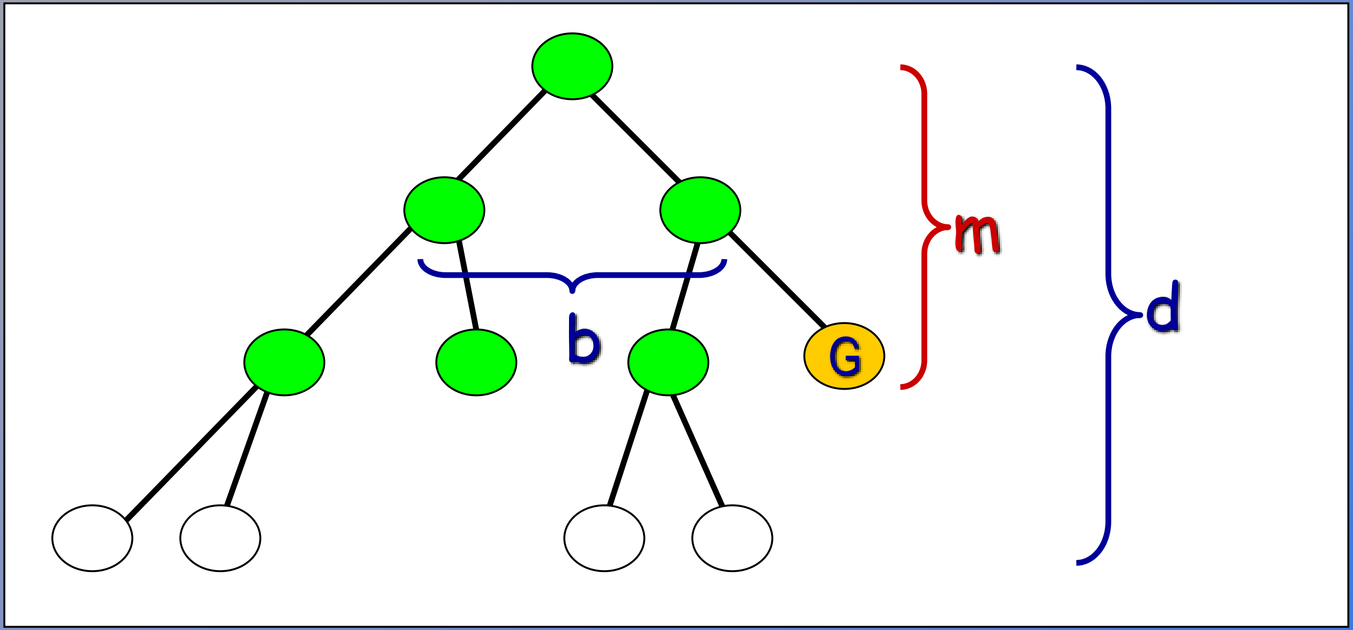
- (S) S removed, (SA,SD) computed and added
- (SA, SD) SA removed, (SAB,SAD,SAS) computed, (SAB,SAD) added
- (SD,SAB,SAD) SD removed, (SDA,SDE,SDS) computed, (SDA,SDE) added
- (SAB,SAD,SDA,SDE) SAB removed, (SABA,SABE,SABC) computed, (SABE,SABC) added
- (SAD,SDA,SDE,SABE,SABC) SAD removed, (SADS,SADA, SADE) computed, (SADE) added
- etc, until QUEUE contains:
- (SABED,SABEF,SADEB,SADEF,SDABC,SDABE,SDEBA,SDEBC, SDEFG) goal is reached: reports success

Completeness (breadth-first)

- COMPLETE
  - ➔ even for infinite implicit NETS !
- Would even remain complete without our loop-checking.
- Note: ALWAYS finds the shortest path.

Speed (breadth-first)

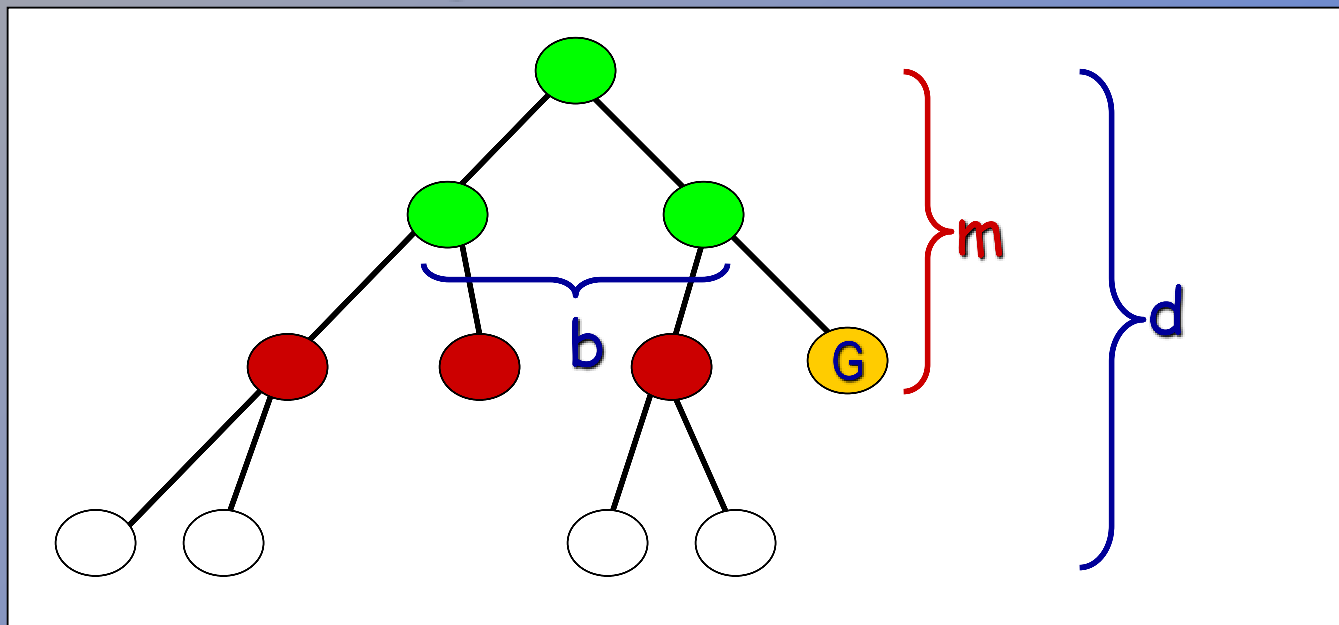
- If a goal node is found on depth m of the tree, all nodes up till that depth are created.

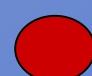


- Thus: $O(b^m)$
- note: depth-first would also visit deeper nodes.

Memory (breadth-first)

- Largest number of nodes in QUEUE is reached on the level m of the goal node.



- QUEUE contains all  and 

 nodes. (Thus: 4) .
- In General: b^m

Practical evaluation:

- Depth-first:
  - ➔IF the search space contains very deep branches without solution, THEN Depth-first may waist much time in them.
- Breadth-first:
  - ➔ Is VERY demanding on memory !
- Solutions ??
  - ➔Non-deterministic search
  - ➔ Iterative deepening

Non-deterministic search:

1. QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO remove the first path from the QUEUE;
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths in random places in QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Iterative deepening search

- Restrict a depth-first search to a fixed depth.
- If no path was found, increase the depth and restart the search.

Depth-limited search:

1. DEPTH <-- <some natural number>
   QUEUE <-- path only containing the root;

2. WHILE QUEUE is not empty
   AND goal is not reached

   DO remove the first path from the QUEUE;
   IF path has length smaller than DEPTH
   create new paths (to all children);
   reject the new paths with loops;
   add the new paths to front of QUEUE;

3. IF goal reached
   THEN success;
   ELSE failure;

Iterative deepening algorithm:

1. DEPTH <-- 1

2. WHILE goal is not reached

   DO { perform Depth-limited search;
        increase DEPTH by 1;

Iterative deepening: the best 'blind' search.

- Complete: yes - even finds the shortest path (like breadth first) .
- Memory: b*m (combines advantages of depth- and breadth-first)
- Speed:
  - ➔ If the path is found for Depth = m, then how much time was waisted constructing the smaller trees??
- $b^{m-1} + b^{m-2} + \ldots + 1 = \dfrac{b^m - 1}{b - 1} = O(b^{m-1})$
- While the work spent at DEPTH = m itself is $O(b^m)$

In general: VERY good trade-off