Uninformed Search

In *implicit graph search*, no graph representation is available at the beginning; only while the search progresses, a partial picture of it evolves from those nodes that are actually explored. In each iteration, a node is *expanded* by generating all adjacent nodes that are reachable via edges of the implicit graph (the possible edges can be described for example by a set of transition rules). This means applying all allowed actions to the state. Nodes that have been generated earlier in the search can be kept track of; however, we have no access to nodes that have not been generated so far. All nodes have to be reached at least once on a path from the initial node through successor generation.

Definition 2.1. *(Closed/Open Lists) The set of already expanded nodes is called* Closed *and the set of generated but yet unexpanded nodes is called* Open. *The latter is also denoted as the search frontier.*

Search Function – Uninformed Searches

```
Open = initial state            // open list is all generated states
                          // that have not been "expanded"
While open not empty            // one iteration of search algorithm
  state = First(open)           // current state is first state in open
  Pop(open)                     // remove new current state from open
  if Goal(state)                // test current state for goal condition
    return "succeed"            // search is complete
                          // else expand the current state by
                          // generating children and
                          // reorder open list per search strategy
  else open = QueueFunction(open, Expand(state))
Return "fail"
```

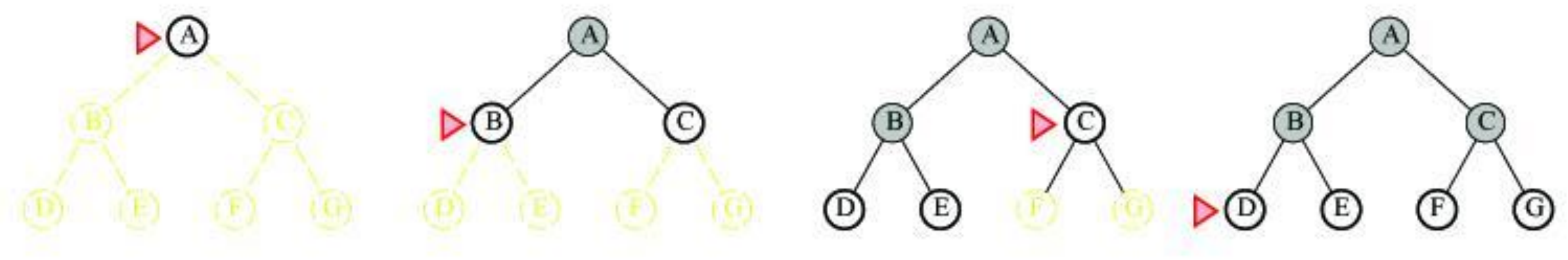
Search Strategies

- Search strategies differ only in Queuing Function
- Features by which to compare search strategies
  - Completeness (always find solution)
  - Cost of search (time and space)
  - Cost of solution, optimal solution
  - Make use of knowledge of the domain
    - “uninformed search” vs. “informed search”

Breadth-First Search

- Generate children of a state, QueueingFn adds the children to the end of the open list
- Level-by-level search
- Order in which children are inserted on open list is arbitrary
- In tree, assume children are considered left-to-right unless specified differently
- Number of children is “branching factor” b

BFS Examples



b = 2

Example

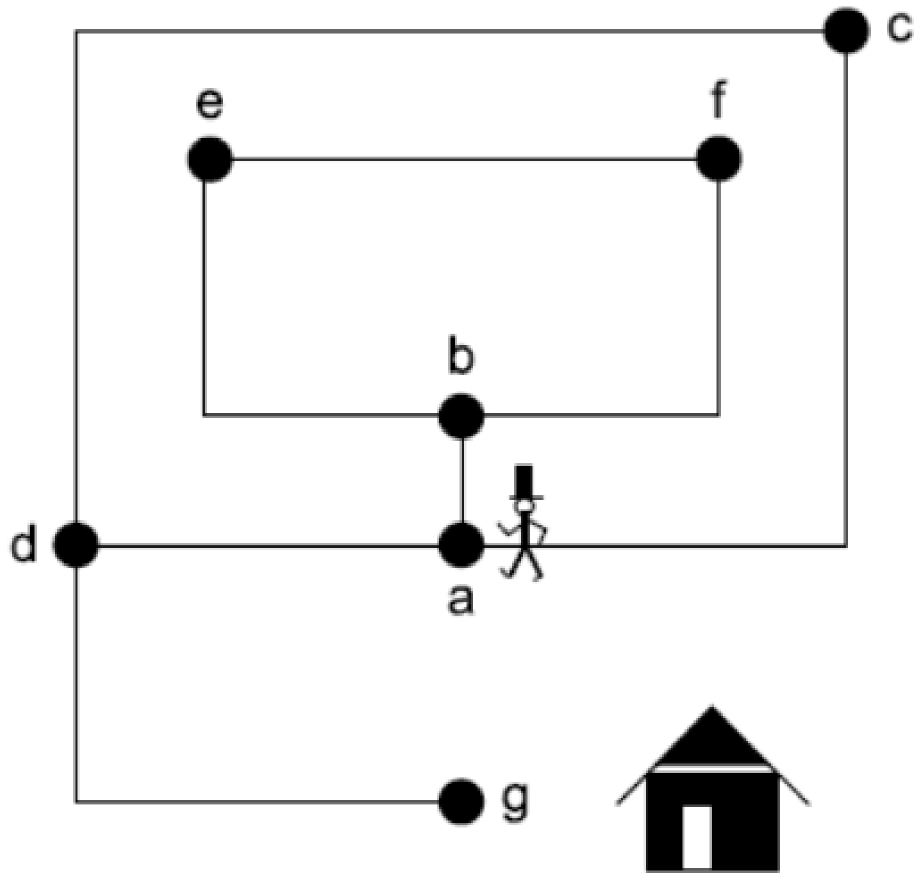


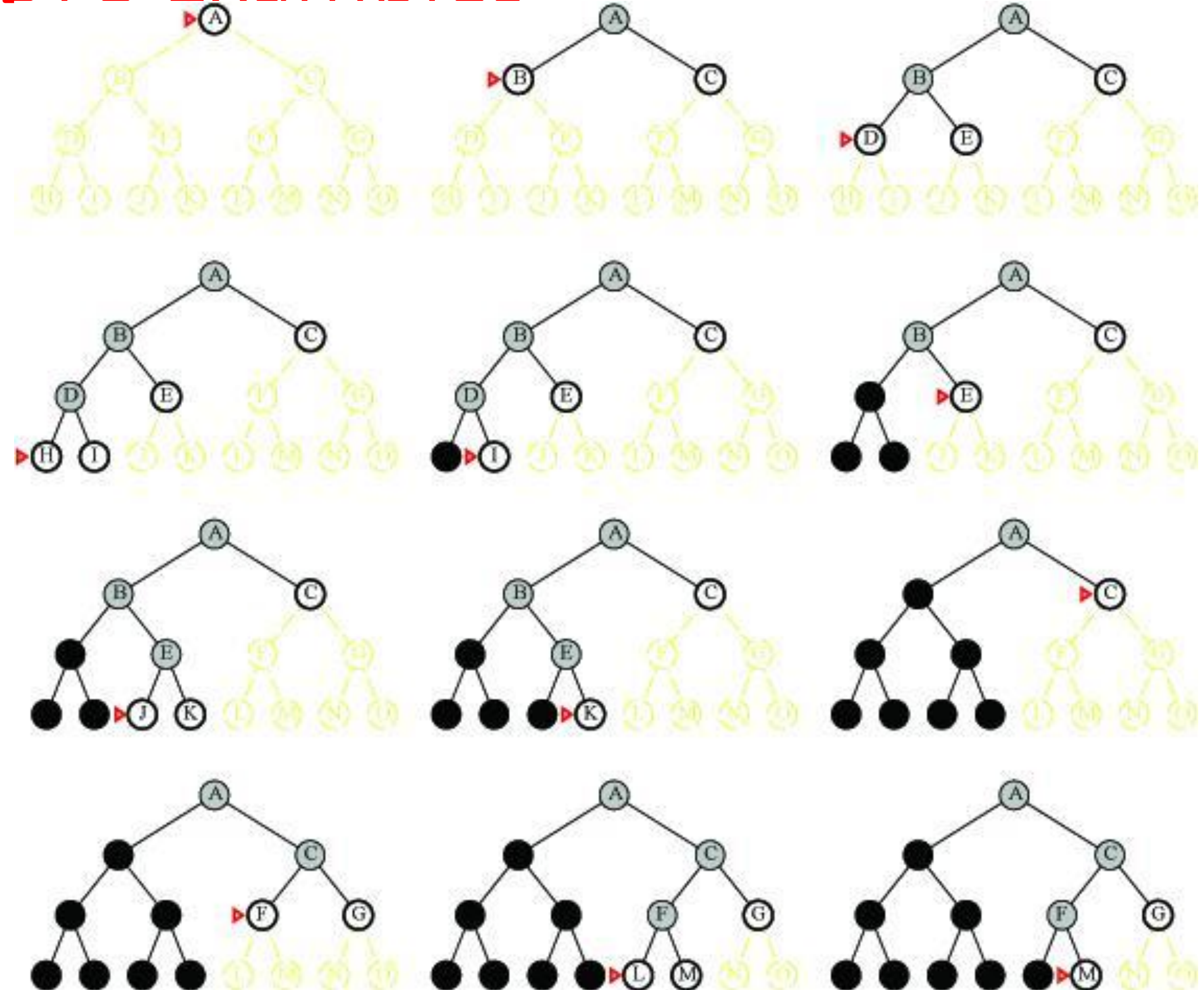
Table 2.2 Steps in BFS (with duplicate detection) for the example in Figure 2.1.

Step	Selection	Open	Closed	Remarks
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{c,d,e,f}	{a,b}	
4	c	{d,e,f}	{a,b,c}	d is duplicate
5	d	{e,f,g}	{a,b,c,d}	c is duplicate
6	e	{f,g}	{a,b,c,d,e}	f is duplicate
7	f	{g}	{a,b,c,d,e,f}	e is duplicate
8	g	{}	{a,b,c,d,e,f,g}	Goal reached

Depth-First Search

- QueueingFn adds the children to the front of the open list
- BFS emulates FIFO queue
- DFS emulates LIFO stack
- Net effect
    - Follow leftmost path to bottom, then backtrack
    - Expand deepest node first

DFS Examples



Example

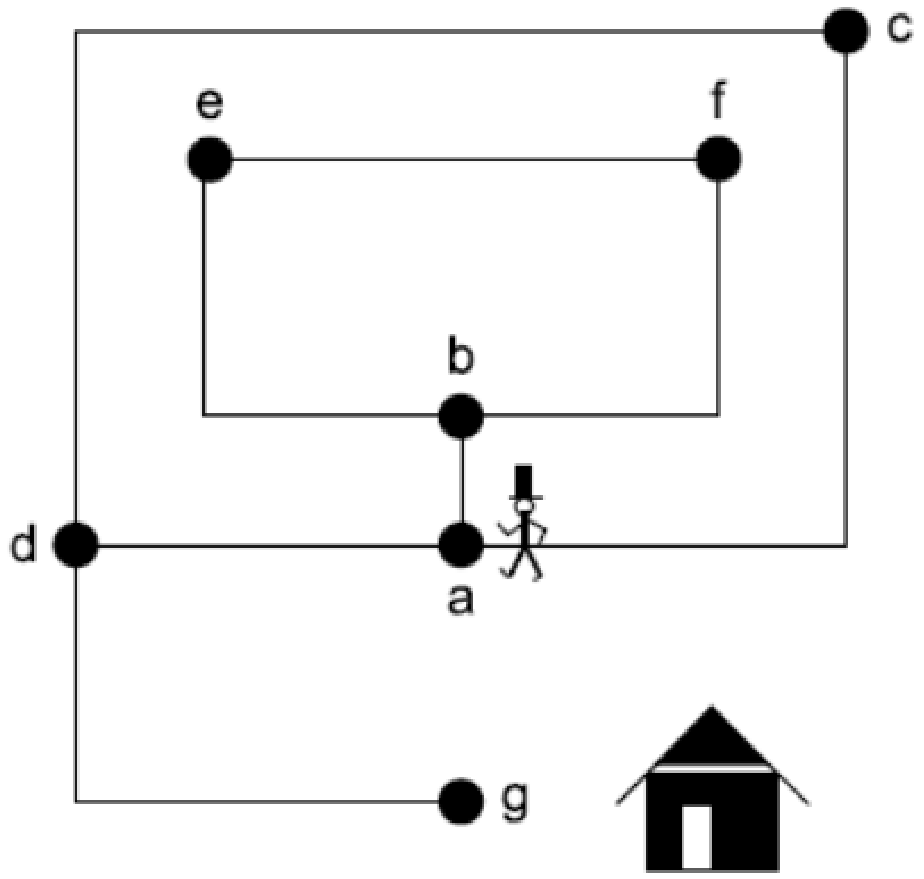


Table 2.1 Steps in DFS (with duplicate detection) for the example in Figure 2.1.

Step	Selection	Open	Closed	Remarks
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{e,f,c,d}	{a,b}	
4	e	{f,c,d}	{a,b,e}	f is duplicate
5	f	{c,d}	{a,b,e,f}	e is duplicate
6	c	{d}	{a,b,e,f,c}	d is duplicate
7	d	{g}	{a,b,e,f,c,d}	c is duplicate
8	g	{}	{a,b,e,f,c,d,g}	Goal reached

2.5.1 Completeness

Both Depth First Search and Breadth First Search are complete for finite state spaces. Both are systematic. They will explore the entire search space before reporting failure. This is because the termination criterion for both is the same. Either they pick the goal node and report success, or they report failure when *OPEN* becomes empty. The only difference is where the new nodes are placed in the *OPEN* list. Since for every node examined, all unseen successors are put in *OPEN*, both searches will end up looking at all reachable nodes before reporting failure. If the state space is infinite, but with finite branching then depth first search may go down an infinite path and not terminate. Breadth First Search, however, will find a solution if there exists one. If there is no solution, both algorithms will not terminate for infinite state spaces.

Time Complexity

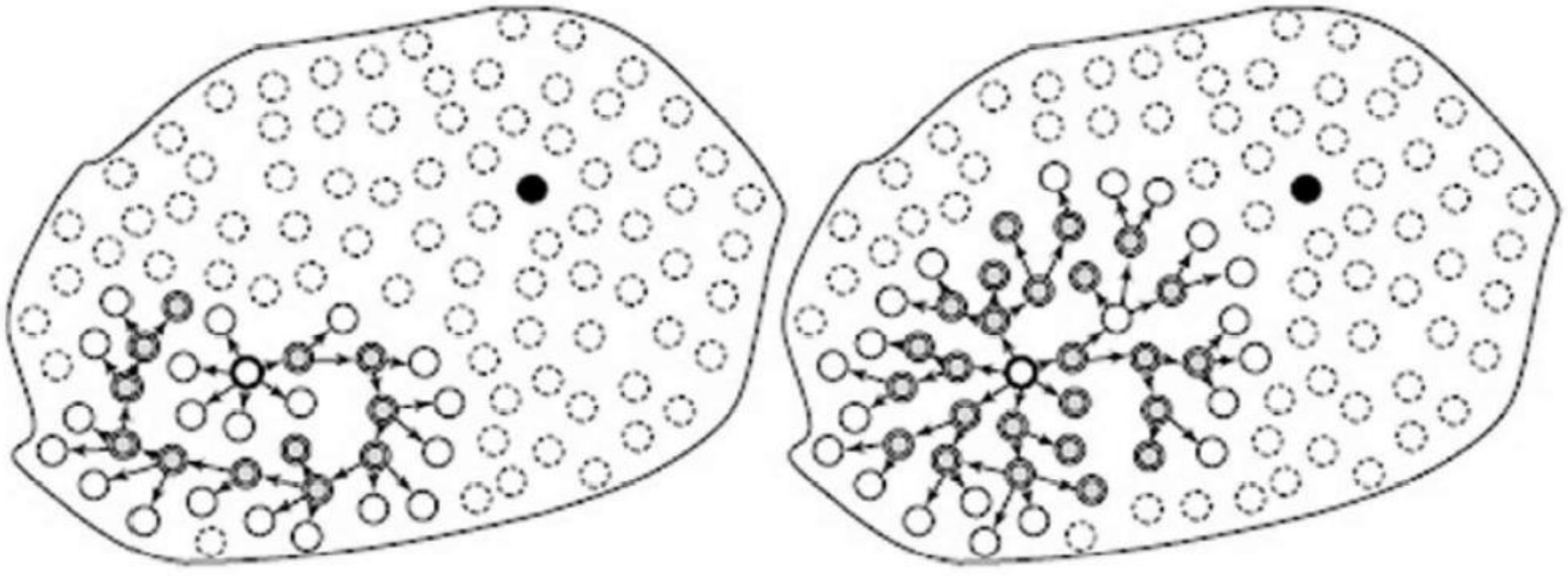


FIGURE 2.23 *DFS* on the left dives down some path. *BFS* on the right pushes slowly into the search space. The nodes in dark grey are in *CLOSED*, and light grey are in *OPEN*. The goal is the node in the centre, but it has no bearing on the search tree generated by *DFS* and *BFS*.

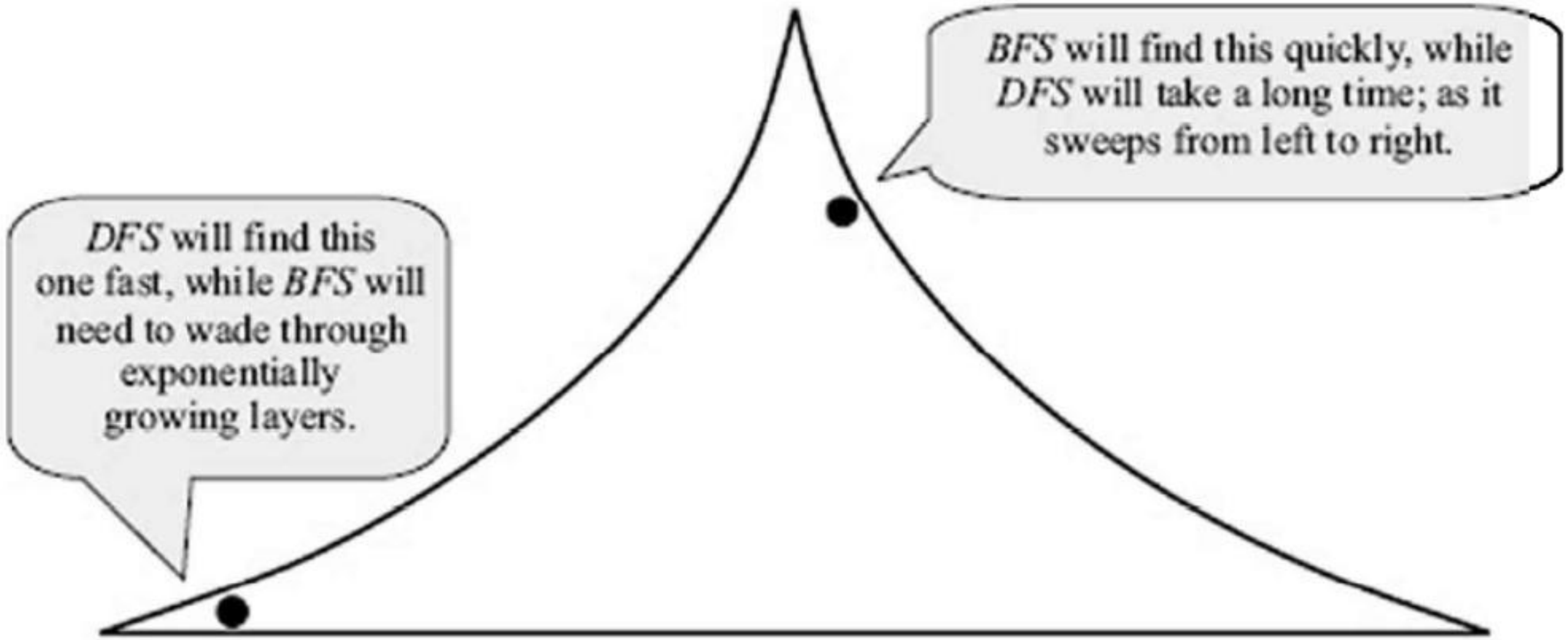

BFS will find this quickly, while *DFS* will take a long time; as it sweeps from left to right.
DFS will find this one fast, while *BFS* will need to wade through exponentially growing layers.

FIGURE 2.24 *DFS* vs. *BFS* on a finite tree.

DFS

If the goal is on the extreme left then *DFS* finds it after examining *d* nodes. These are the ancestors of the goal node. If the goal is on the extreme right, it has to examine the entire search tree, which is b^d nodes. Thus, on the average, it will examine N_{DFS} nodes given by,

$$\begin{aligned} N_{DFS} &= [(d+1) + (b^{d+1} - 1)/(b-1)] / 2 \\ &= (b^{d+1} + bd + b - d - 2) / 2(b-1) \\ &\approx b^d/2 \text{ for large } d \end{aligned}$$

$$I = 1 + b + b^2 + \cdots + b^{d-1}$$

$$I = \frac{(b^d - 1)}{(b-1)}$$

BFS

The search arrives at level *d* after examining the entire subtree above it. If the goal is on the left, it picks only one node, which is the goal. If the goal is on the right, it picks it up in the end (like the *DFS),* that is, it picks b^d nodes at the level *d*. On an average, the number of nodes N_{BFS} examined by *BFS* is

$$\begin{aligned} N_{BFS} &= (b^d - 1)/(b-1) + (1 + b^d)/2 \\ &\approx (b^{d+1} + b^d + b - 3) / 2(b-1) \\ &\approx b^d (b-1) / 2b \text{ for large } d \end{aligned}$$

2.5.3 Space Complexity

For assessing space complexity, we analyse the size of the *OPEN* list. That is, the number of candidates that the search keeps pending for examination in the future.

DFS

In a general search tree, with a branching factor *b*, *DFS* dives down some branch, at each level, leaving (b – 1) nodes in *OPEN*. When it enters the depth *d*, it has at most O_{DFS} nodes in the *OPEN* list, where

$$\begin{aligned} O_{\mathrm{DFS}} &= (b-1)(d-1)+b \\ &= d(b-1)+1 \end{aligned}$$

BFS

Breadth First Search pushes into the tree level by level. As it enters each level at depth *d*, it sees all the b^d nodes ahead of it in *OPEN*. By the time it finishes with these nodes, it has generated the entire next level and stored them in *OPEN*. Thus, when it enters the next level at depth (d + 1), it sees $b^{(d+1)}$ nodes in the *OPEN* list.

Analysis

- See what happens with b=10
  - expand 10,000 nodes/second
  - 1,000 bytes/node

Depth	Nodes	Time	Memory
2	1110	.11 seconds	1 megabyte
4	111,100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
15	10^{15}	3,523 years	1 exabyte

DFS requires 118kb instead of 10 petabytes for d=12 (10 billion times less)

Comparison of Search Techniques

	DFS	BFS
Complete	N	Y
Optimal	N	N
Heuristic	N	N
Time	b^m	b^{d+1}
Space	bm	b^{d+1}