

# CSC384:Lecture4

## ■ Lasttime

- done with DCL (except for the fact that we'll **use** it!)

## ■ Today

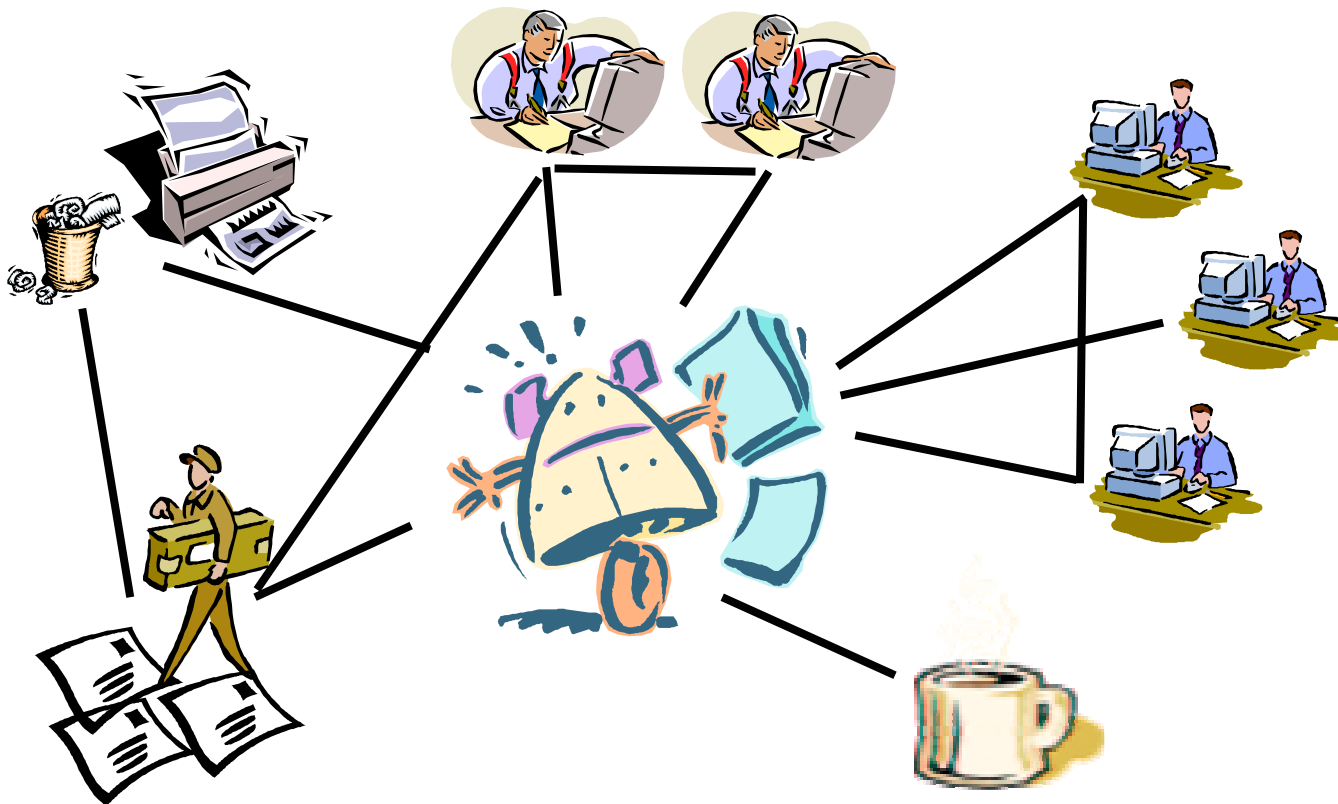
- quick summary of uses of DCL (from last time)
- Intro to search; generic search procedure; BrFS, DFS, path extraction, cycle and mult. path checking

## ■ Readings:

- Today: Ch. 4.1 – 4.4, 4.6
- Next week: Ch. 4.5/4.6

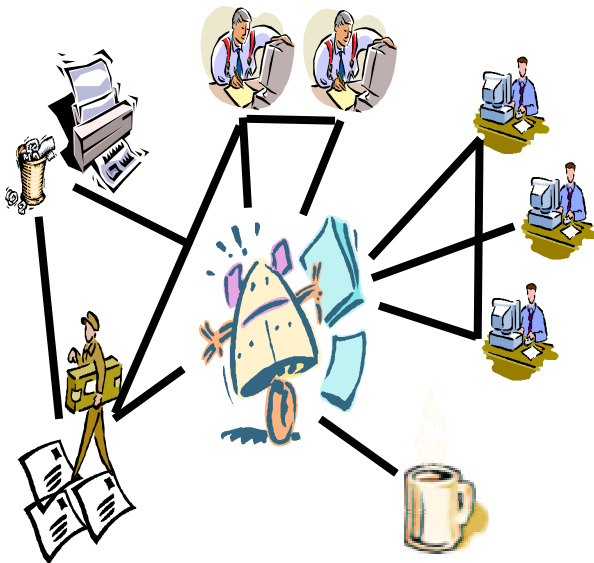
# A Planning Problem

- A planning problem: we want robot to decide what to do; how to act to achieve our goals



# A Planning Problem

- How to *change* the world to suit our needs
- Critical issue: we need to reason about *how the world will be* after doing a few actions, not *just* what it is like now



**GOAL:** Craig has coffee

**CURRENTLY:** robot in mailroom,  
has no coffee, coffee not made,  
Craig in office, etc.

TO DO: goto lounge, make coffee,...

# Planning

- So far, we've seen DCL used to reason about *static environments* (what the world *is like*)
  - what is correct treatment for symptom X?
  - is this region water or land?
- Want to use DCL to reason about *dynamic environments*
  - A,B,C are true: will they still be true after doing X?
  - A,B,C are true: what do I need to do to make D true?
- The heart of decision making!
  - complexities: uncertainty (where is craig? navigate stairs?); *many actions to choose from (what is right sequence?)*; exogenous events (battery loses charge)

# Graph Search

- We'll abstract away complexities of planning
  - focus on **finding the right sequence of actions** only
  - going to ignore the fact that at any point in time many things are true and false
- Treat planning as the **search for a path** from one state (of the world) to a desired goal state
- **Informally:** We have a set of **states**, and a set of **moves**/actions that take us from one state to another. Given an **initial state** (current) and **target state** (goal), find a **sequence of moves** that gets me from initial state to goal
  - or *shortest* sequence, or *cheapest* sequence, or ...

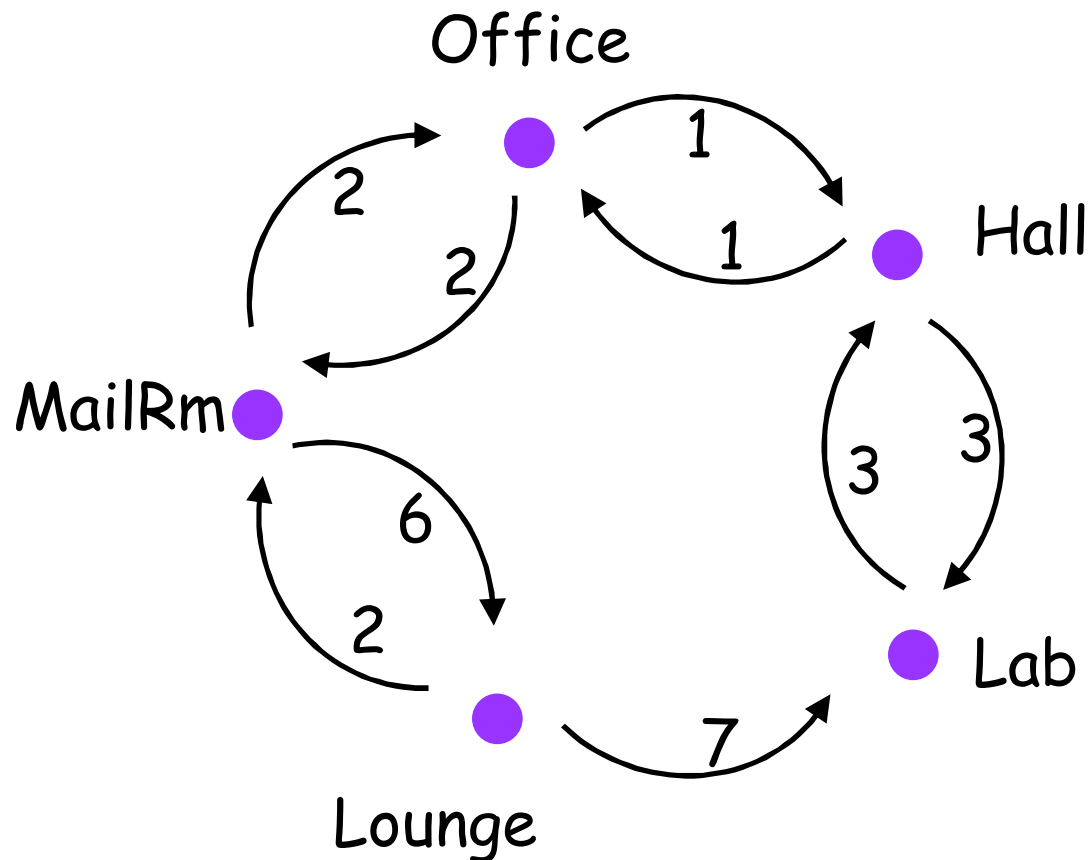
# Graph Search is Very General

- This viewpoint applies to a wide variety of tasks
  - **RoboCof** – find plan that gives Craig coffee; route across floor plan; etc. [states? moves? objective?]
  - **Games** – 8-puzzle; backgammon; chess; Doom; [complications: other players making moves]
  - Scheduling, logistics, planning, most optimization problems
  - **Medical diagnosis**
  - **Scene interpretation** – find a consistent labeling
  - **Finding a derivation** in DCL/Prolog [what are states? moves? goal?]
  - **Finding a good travel package** [states? moves?]
  - Almost every problem in AI can be viewed this way!

# Graph-based Search Formalization

- A directed graph: set of nodes  $N$ , set of directed edges  $E \subseteq N \times N$  (these are *ordered* pairs)
  - nodes correspond to states we can move among
- If  $\langle n_1, n_2 \rangle \in E$ , then  $n_2$  is a *neighbor* of  $n_1$ 
  - written  $n_1 \rightarrow n_2$  (relation is not symmetric)
  - edges correspond to possible moves we can make
- A *node labeling* is a function  $L_n: N \rightarrow Nlabels$ 
  - denote properties of states (e.g., a value)
- An *edge labeling* is a function  $L_e: E \rightarrow Elabels$ 
  - denote properties of edges (e.g., a move cost)

# An Very Simple Search Graph



Nodes (states) are locations robot can move among

Edges are routes among these locations

Edge labels denote costs (e.g., expected travel time)



# Paths and Cycles

- A *path* in graph  $G = (N, E)$  is a sequence of nodes  $\langle n_1, n_2, \dots, n_k \rangle$  s.t. each  $\langle n_i, n_{i+1} \rangle \in E$ 
  - $n_1 \rightarrow n_2 \rightarrow \dots \rightarrow n_k$  is a *path from  $n_1$  to  $n_k$*
- A *cycle* is a path  $\langle n_1, n_2, \dots, n_k \rangle$  with  $n_1 = n_k$  ( $k > 1$ )
  - A graph  $G$  is acyclic if no path in  $G$  is a cycle
  - A path is acyclic if no subpath is a cycle

# Graph Search Problems

- A **graph search problem**: given graph  $(N, E)$ , a start node  $s \in N$ , and a set of goal nodes  $G \subseteq N$ , find a path  $P$  from  $s$  to some  $g \in G$  satisfying property  $X$ .
- Possible properties  $X$ :
  - $X = \text{"null"}$ : Any path to any node in  $G$  will do
    - *Find me any path to the office*
  - $X = \text{"P is the best path to goal G"}$ : this means we have some optimization criterion we need to satisfy
    - possible criteria: shortest (# edges); least cost; etc.
    - *Find me shortest/fastest path to office*
  - $X = \text{"P has quality } \geq q \text{"}$  (a satisficing problem)
    - *Find me a path that gets me to the office by 10AM*

# A Couple Notes

- If arcs are labeled with actions, we often want to return the sequence of actions, not just the path
- Most interesting search problems involve *implicit search graphs*
  - e.g., consider chess: nobody constructs a graph of all  $10^{30}$  board positions explicitly
  - e.g., Prolog: answer clauses generated as derived
  - when solving search problem, we generate neighbors as we need them
  - define states, neighborhood relation...or define moves and how to generate neighboring state

# Generic Search Procedure

- Many graph search techniques share the following common structure
  - let the **frontier** refer to set of nodes we already know how to reach from the start node  $s$

1. Let  $F = \{s\}$  ;; (initial frontier is start node)
2. Loop until frontier is empty
  - (a) Choose some node  $n$  on frontier; remove  $n$  from  $F$
  - (b) If  $n \in G$  then stop; report success
  - (c) Otherwise add each neighbor of  $n$  to  $F$

# A Generic DCL Implementation

`search( F ) :- select(Node, F, RemF), is_goal(Node).`

`search( F ) :- select(Node, F, RemF),  
                  nbs(Node, NbList),  
                  add_to_frontier(RemF, NbList, NewF),  
                  search( NewF ).`

1. `nbs(Node, NbList)` defines the search graph
  - for each node, we assert its list of nbs in KB
  - if implicit search graph, nbs will generate NBLIST
2. `is_goal(N)` defines the set of goal nodes
  - for each goal node, we assert this in KB

# Generic Search Procedure

- Assume frontier represented as a list of nodes
  - $search(F)$  returns yes there is some path to  $G$  from  $F$
  - so call with start node  $s$  on frontier:  $search([s])$
- $select(N, F, RemF)$ : true if selecting node  $N$  from frontier  $F$  leaves remaining frontier  $RemF$
- $add\_to\_frontier(RemF, NbList, NewF)$ : true if adding nodes in  $NbList$  to  $RemF$  results in  $NewF$

# One Instantiation of Search Proc.

```
search( F ) :- select(Node, F, RemF), is_goal(Node).  
search( F ) :- select(Node, F, RemF), nbs(Node, NbList),  
                add_to_frontier(RemF, NbList, NewF),  
                search( NewF ).
```

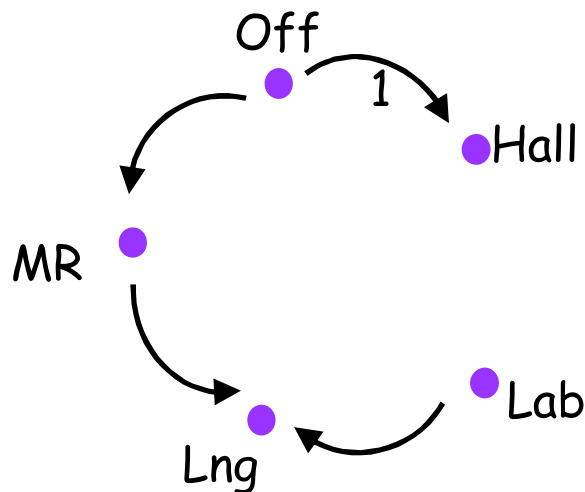
`select(Node, [Node|RemF], RemF).`

`add_tf (RemF, NbList, NewF) :-  
 append(NbList, RemF, NewF).`

## ■ Simple implementation

- you can only select the first node on the frontier
- you add neighbors of selected node to the beginning of the frontier (just append them to front)

# A Modified Example



## Graph:

nb(off,[mr,h]).  
nb(h, []).  
nb(mr, [lng]).  
nb(lng, []).  
nb(lab, [lng]).

## Goal Node:

isgoal(h).

## Start Node:

office

search([off]).

*select off, RF = []*

*not a goal, NF = [mr, h]*

search([mr, h]).

*select mr, RF = [h]*

*not a goal, NF = [lng, h]*

search([lng]).

*select off, RF = [h]*

*not a goal, NF = [h]*

search([h]).

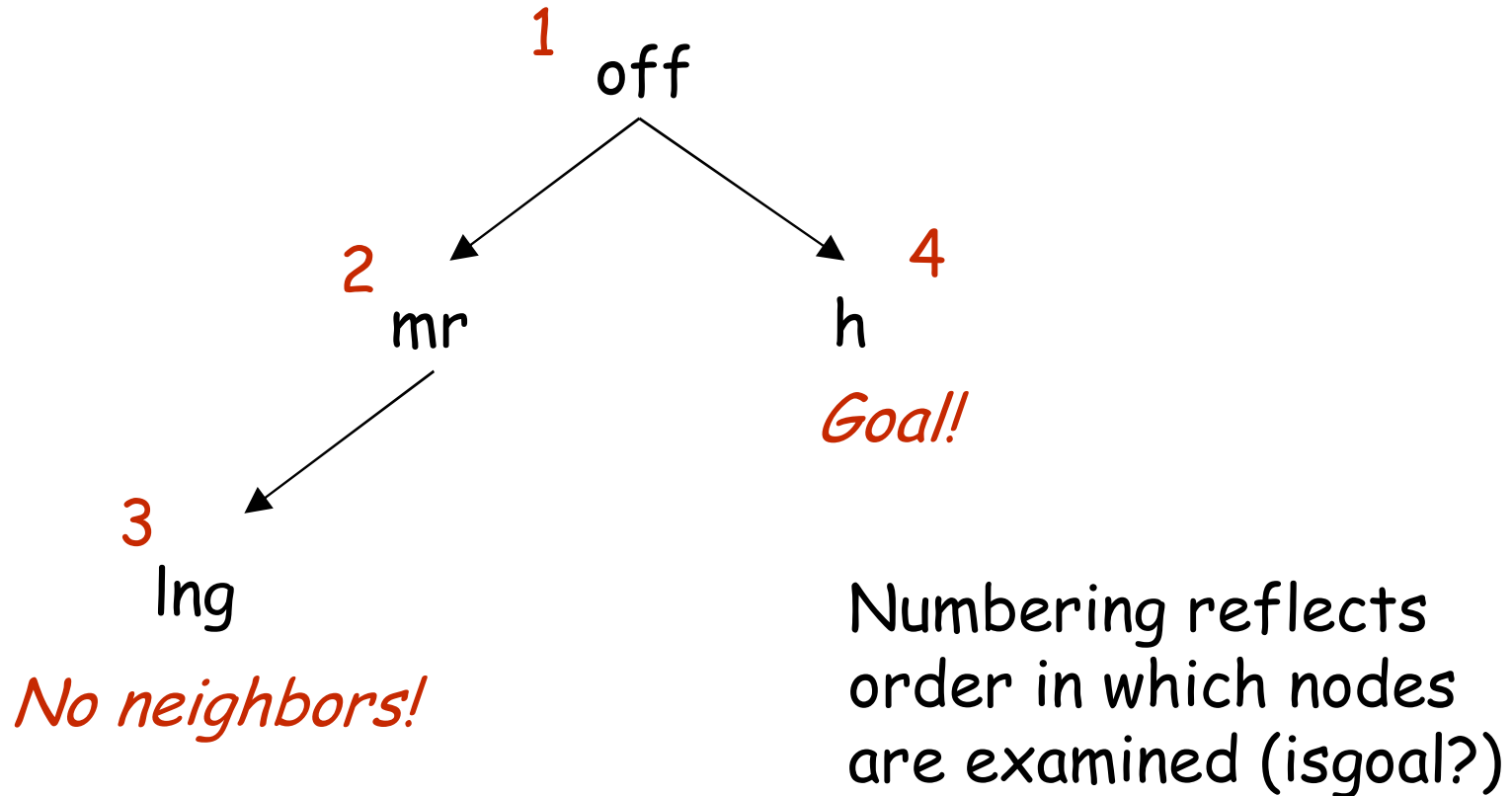
*select off, RF = []*

*is a goal, NF = [mr, h]*

Return yes!



# Search Tree for Modified Example



*What if we added an edge from Ing to lab?*

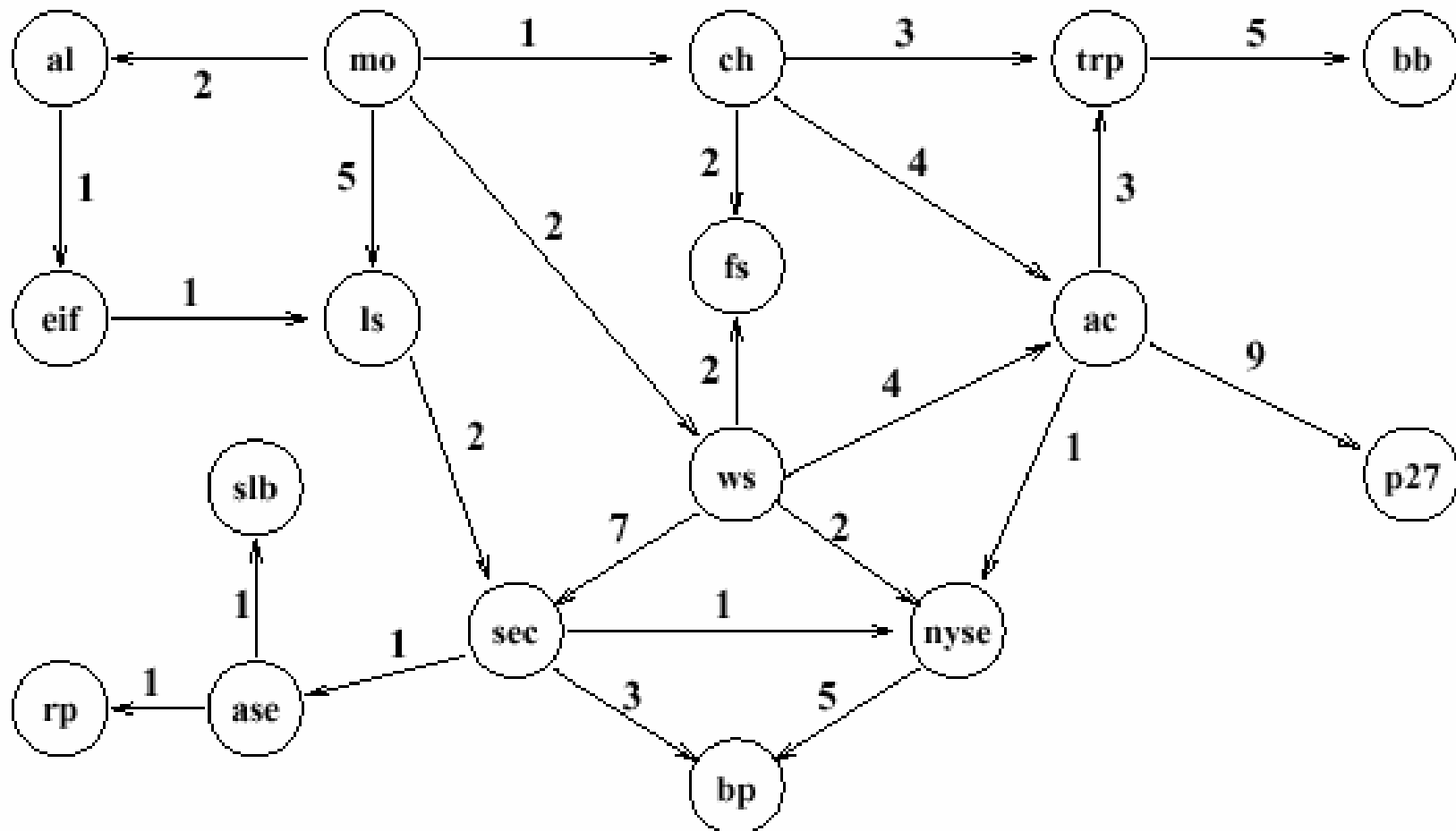
# Depth-First Search

- The specific instantiation we've seen is simply **depth-first search** (which you've seen previously, no doubt)
  - in the search tree, we work deep into the tree until we reach a dead-end, then we “backtrack”
- In generic search algorithm, this is achieved by:
  - always selecting first node on the frontier
  - always inserting the new neighbors at head of frontier
- Note: all “bookkeeping” required for backtracking is taken care of by organization of the frontier

# Defn of Search Tree

- Given a search graph, start node  $s$
- The *search tree rooted at  $s$*  is a tree such that:
  - root node is  $s$
  - each node has all its neighbors for children
- So at level  $k$  of the tree are all states that you can reach in exactly  $k$  moves (where root = 0)
  - each path in search graph (including cyclic paths) is a path through search tree
  - nodes in graph can appear *many* times in tree
- Frontier moves “down” the tree

# Manhattan Bike Courier (Acyclic)

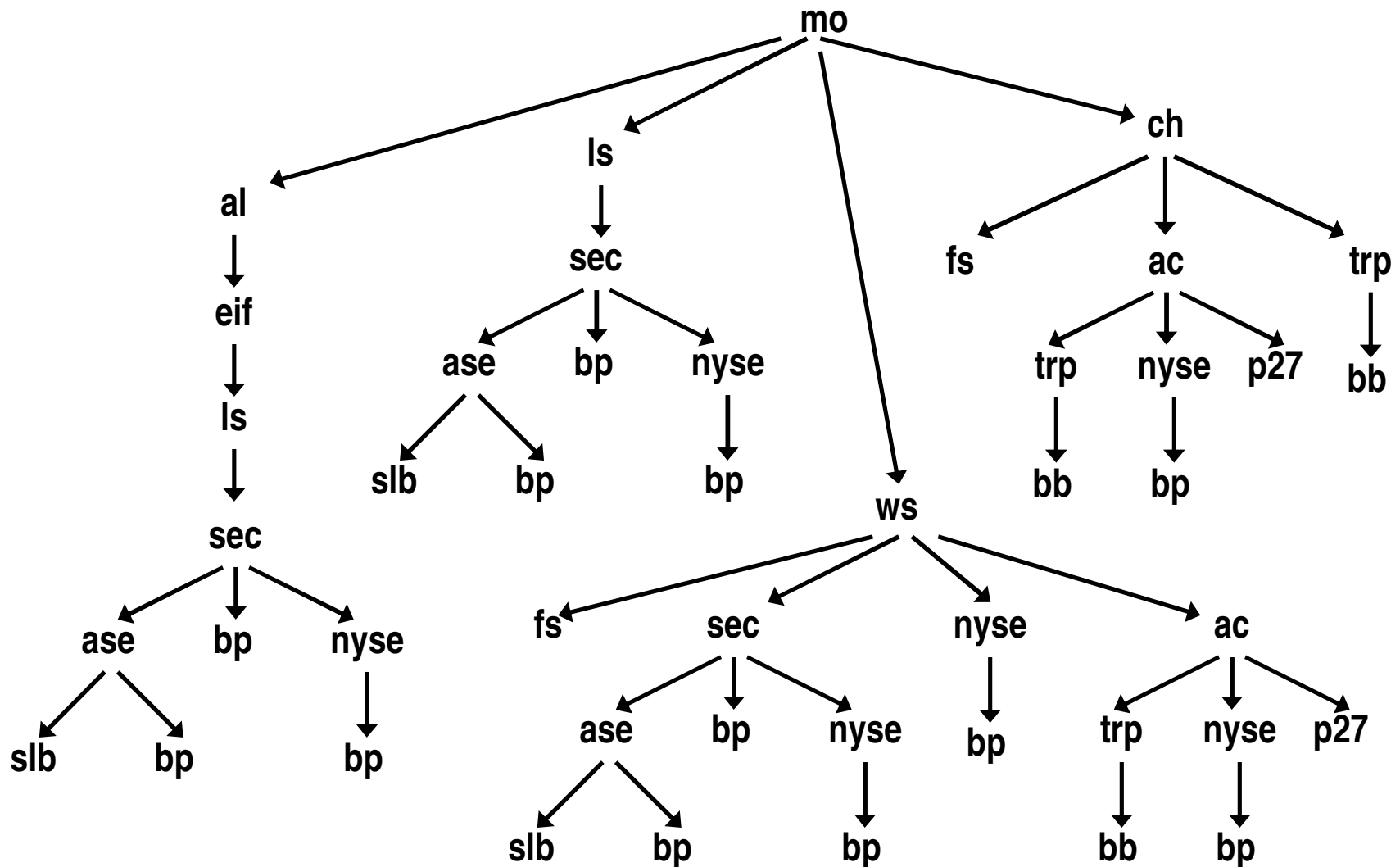


# The MBC Rep'n (No Costs)

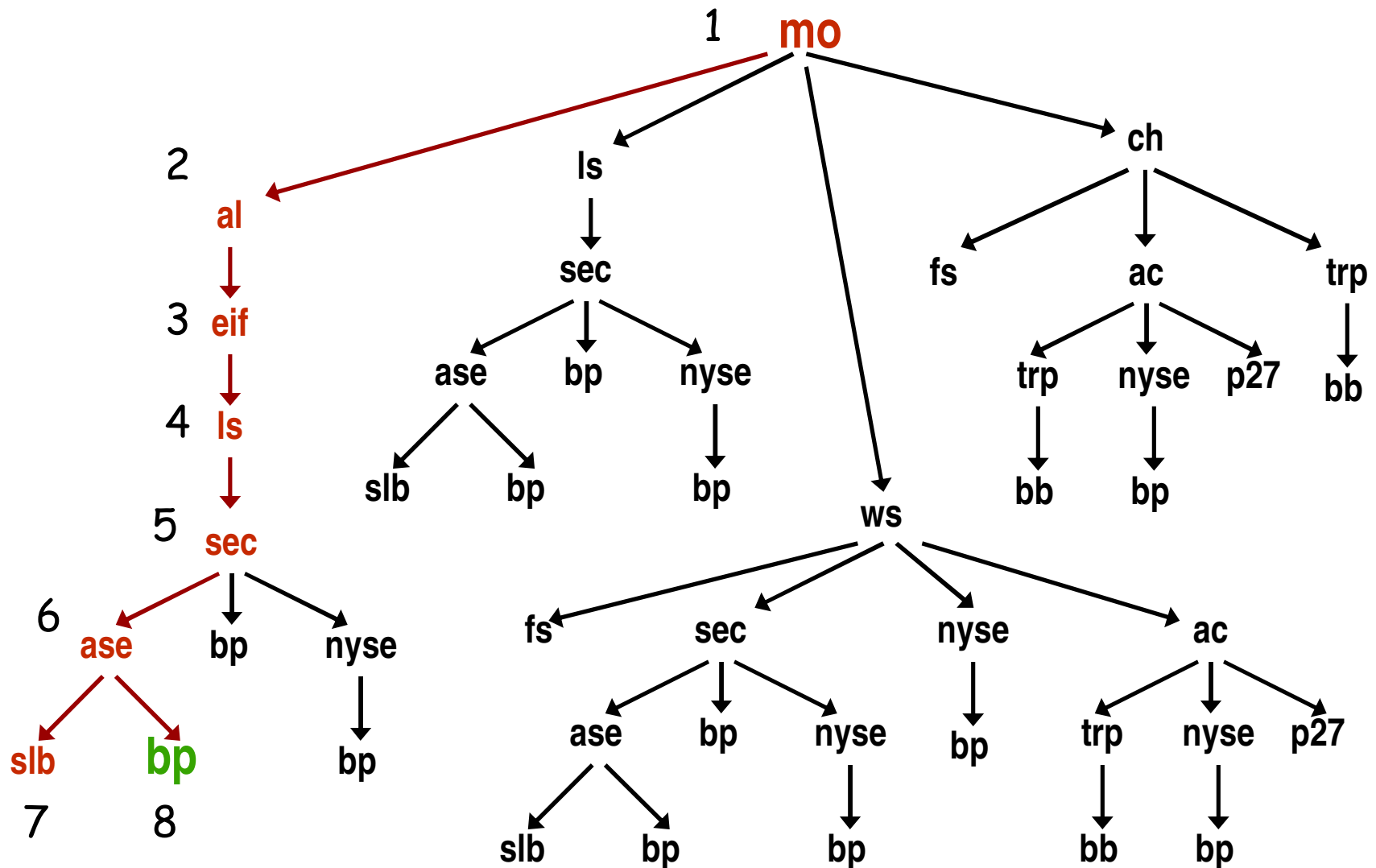
mo: Main Office	ch: City Hall	trp: T.Rowe Price
al: Alley	fs: Fulton Street	bb: Brooklyn Bridge
EIF: Ellis Island Ferry	ws: Wall Street	ac: Anderson Consulting
ls: Loeb Securities	sec: Securities Exch. Comm.	p27: Pier 27
slb: Shearson-Lehmann Bros.	bp: Battery Park	nyse: NY Stock Exch.
rp: Rector Park	ase: Amer. Stock Exch.	

```
nb(mo, [al, ls, ws, ch]).      nb(slb, []).
nb(sec, [ase, bp, nyse]).      nb(fs, []).
nb(ws, [fs, sec, nyse, ac]).   nb(eif, [ls]).
nb(ac, [trp, nyse, p27]).      nb(al, [eif]).
nb(ase, [slb, rp]).            nb(ls, [sec]).
nb(ch, [fs, ac, trp]).         nb(rp, []).
nb(trp, [bb]).                 nb(p27, []).
nb(bb, []).                    nb(bp, []).
nb(nyse, [bp]).
```

# Search Tree: MBC Acyclic; Start *mo*



# DepthFirst Search: Start *mo*; Goal *bp*



# Example Summary

- DFS *expands* eight nodes in this example
  - it examines eight nodes and tests if they are the goal (if they are not, it add neighbors to frontier)
- DFS does not find the shortest path to bp
  - the order in which it is required to examine nodes doesn't allow it to find the shortest path



# Path Extraction

```
search( F ) :- select(Node, F, RemF), is_goal(Node).  
search( F ) :- select(Node, F, RemF), nbs(Node, NbList),  
                add_to_frontier(RemF, NbList, NewF),  
                search( NewF ).
```

- Generic search procedure behaves as follows:
  - assert *isgoal(bp)* ; ask *?search([mo])*.
  - algorithm says yes.
  - but what path does courier take??
  - says yes if there is a path, but doesn't tell us what it is
- We need a *path extraction mechanism*
  - simplest thing to do is store paths on frontier
  - when you select a node from frontier, you also have a specific path to that node attached
- *search(F,Path)* : true if Path is solution to search

# Implementing Path Extraction

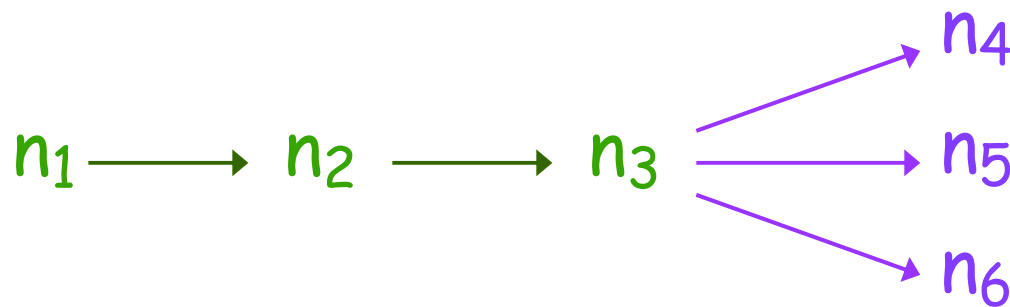
Assume a path is represented as a list of nodes in reverse order: so the path  $mo \rightarrow al \rightarrow eif$  is represented as the list:  $[eif, al, mo]$

```
search( F, [Node | RestP] ) :-  
    select( [Node | RestP] , F, RemF ),  
    is_goal(Node).
```

```
search( F, Path ) :-  
    select( [Node | RestP] , F, RemF ),  
    nbs(Node, NbList),  
    extendpath(NbList, [Node | RestP], NewPaths ),  
    add_to_frontier(RemF, NewPaths, NewF),  
    search( NewF, Path ).
```

# Notes on Path Extraction

- We call search from start node *mo* using *search( [ [mo] ], Path )*.
  - initial frontier consists of a length one *path* (not node)
- Only new predicate needed is *extendpath*
  - basic idea: given a path  $[n_3, n_2, n_1]$  and neighbors of  $n_3$  specified by  $nbs(n_3, [n_4, n_5, n_6])$ ; it produces 3 new paths organized in a list:  
 $[ [n_4, n_3, n_2, n_1] , [n_5, n_3, n_2, n_1] , [n_6, n_3, n_2, n_1] ]$



# DFS with Path Extraction

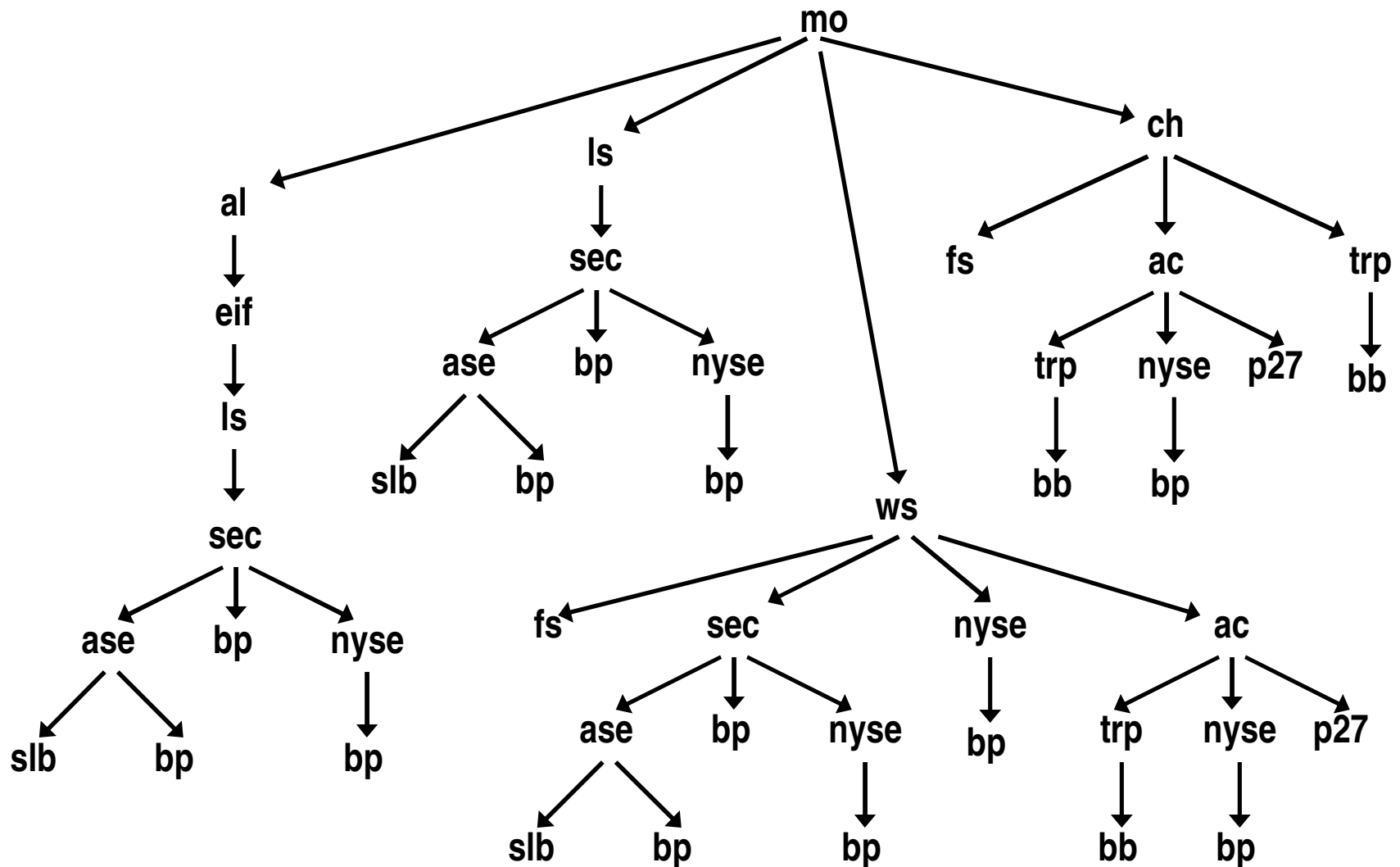
```
search( F, [Node | RestP] ) :-  
    select([Node | RestP], F, RemF),  
    is_goal(Node).
```

```
search( F, Path ) :-  
    select( [Node | RestP], F, RemF ),  
    nbs(Node, NbList),  
    extendpath(NbList, [Node | RestP], NewPaths ),  
    add_to_frontier(RemF, NewPaths, NewF),  
    search( NewF, Path ).
```

```
select(Path, [Path|RestPaths], RestPaths).
```

```
add_tf (RemF, Paths, NewF) :-  
    append(Paths, RemF, NewF).
```

# Search Tree: MBC Acyclic; Start *mo*



# Trace of DFS (with paths: *mo* to *fs*)

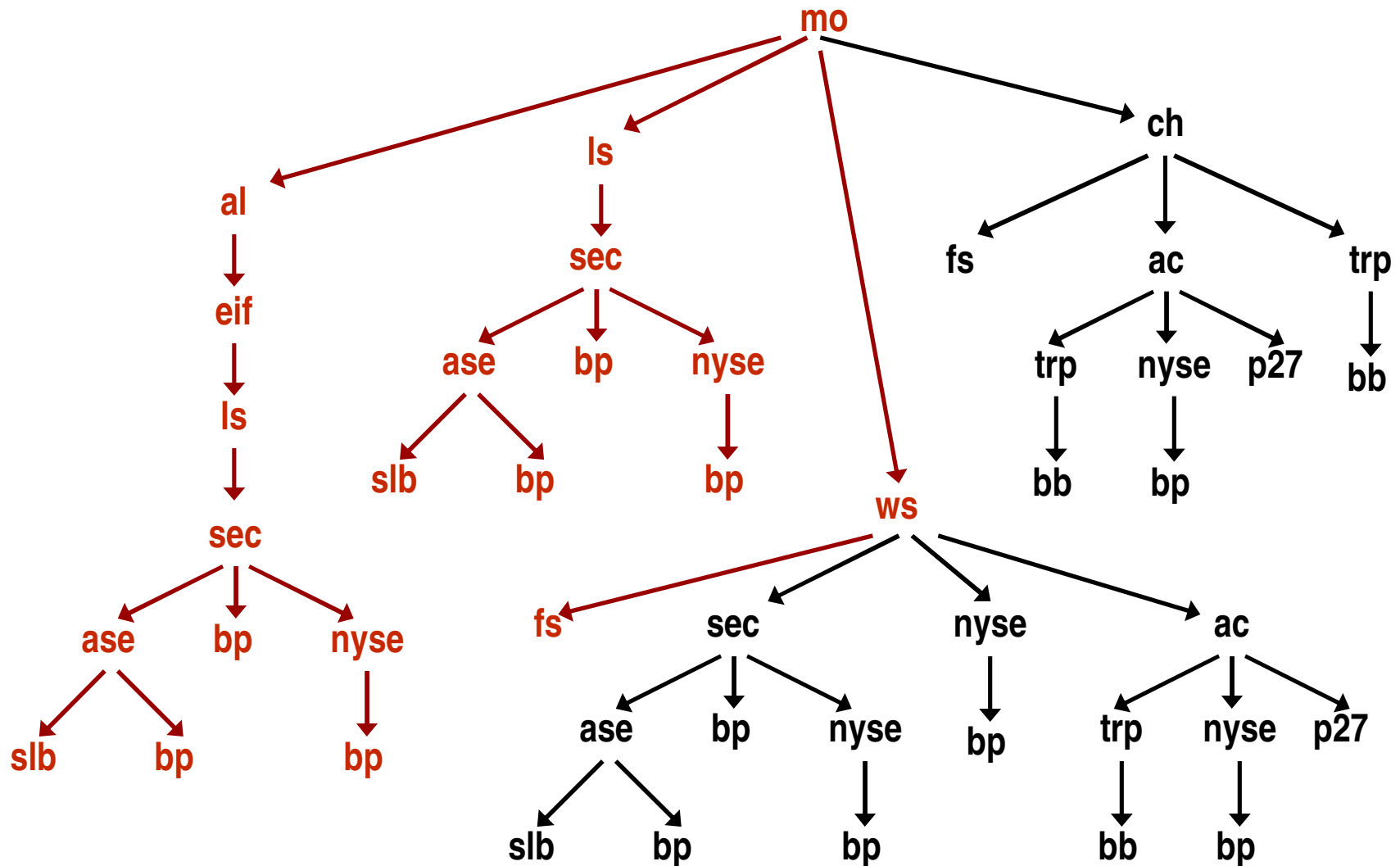
## Frontier evolution:

1. [mo]
2. [al,mo] [ls,mo] [ws,mo] [ch,mo] (= A)
3. [eif,al,mo] ..A..
4. [ls,eif,al,mo] ..A..
5. [sec,ls,eif,al,mo] ..A..
6. [ase,sec,...,mo]  
[bp,sec,...,mo] [nyse,sec,...,mo] ..A.. (= B)
7. [slb,ase,...,mo] [rp,ase,...,mo] ..B.. ..A..
8. [rp,ase,...,mo] ..B.. ..A..
9. [bp,sec,...,mo] [nyse,sec,...,mo] ..A..
10. [nyse,sec,...,mo] ..A..

11. [bp,nyse,sec,...,mo] ..A..
  12. [ls,mo] [ws,mo] [ch,mo]
  13. [sec,ls,mo] [ws,mo] [ch,mo]
  - 14-19. Exactly like expansion of  
[sec,ls,wtc,al,mo] in Steps 6-11
  20. [ws,mo] [ch,mo]
  21. [fs,ws,mo] [sec,ws,mo]  
[nyse,ws,mo] [ac,ws,mo] [ch,mo]
- GOAL = fs is found

Total Search steps: 21 nodes expanded
--

# Paths Explored by DFS in Example

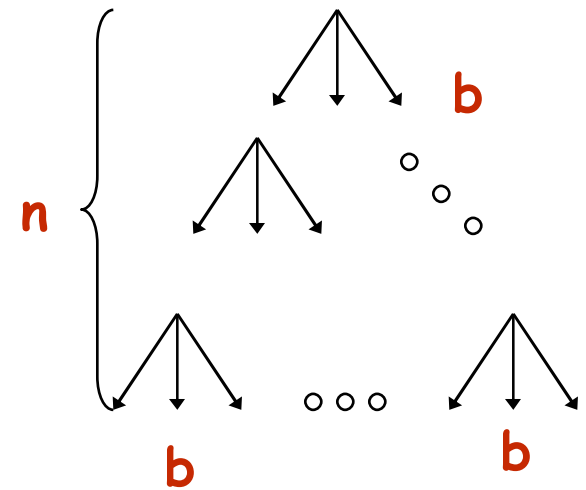


# Properties of DFS: Time

## ■ How long can DFS take?

### (a) Finite graph, no cycles:

- Could explore each branch of search tree (until goal found or search fails).
- If branching factor bounded by  $b$ , depth bounded by  $n$ , then we explore  $O(b^n)$  full paths (length  $n$ )
- note:  $n \leq N$  (number of nodes in  $G$ )



### (b) Finite graph with cycles:

- may not terminate unless we perform cycle checking (later)
- Does it ever make sense to explore a cyclic path?

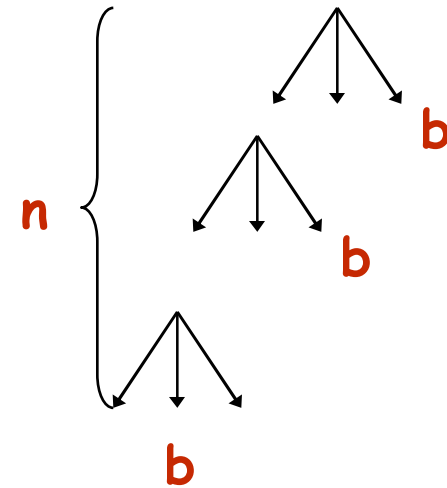
### (c) If we're **lucky** with the node ordering:

- may find soln in  $n^*$  steps ( $n^*$  is length of shortest path to goal)



# Properties of DFS: Space

- How many paths on frontier at any one time?
  - If current path length  $m$  (i.e., current node selected), then there are  $bm$  paths on frontier
  - If longest path is length  $n$ , then never more than  $bn$  paths
  - $b$  paths have length 1,  $b$  length 2, etc. up to  $b$  paths of length  $n$
  - Total space  $b + 2b + 3b + \dots nb$  which is  $O(n^2b)$  space
- Is quadratic space required?
  - No: many paths have common substructure
  - Consider how to store frontier in linear space  $O(nb)$ : use a tree!



# Properties of DFS: Solution Quality

- Will DFS find the shortest (min # arcs) solution?
  - In general, no.
  - It can if you are lucky.
  - In Manhattan (acyclic) problem, with start mo and goal ls, it returns soln:  $mo \rightarrow al \rightarrow eif \rightarrow ls$ , even though shorter solution  $mo \rightarrow ls$  exists.
- So how can we find shortest path?

# Breadth-First Search

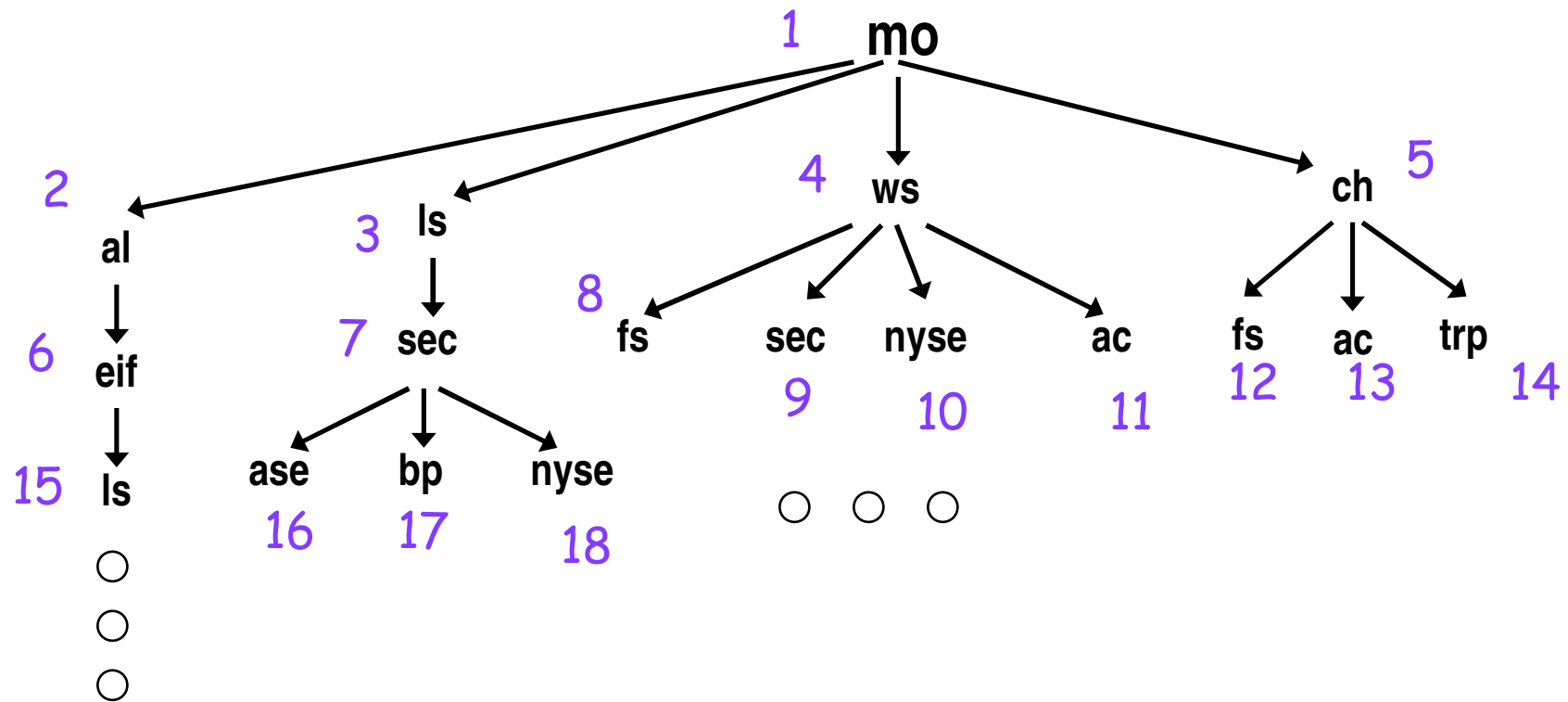
- One way to ensure shortest solution is found (wrt # of arcs) is to explore paths in order of length
- **Breadth-first search (BFS)** does exactly this
- It is implemented by selecting nodes/paths from the front of the frontier (like DFS), but inserting new neighbors/paths at the *end of the frontier*

`select(Path, [Path|RestPaths], RestPaths).`

`add_tf (RemF, Paths, NewF) :-  
 append( RemF, Paths, NewF).`

differs from DFS  
only in *order* of  
arguments to  
append

# Frontier Growth in BFS



etc.

# Trace of BFS (with paths: *mo* to *fs*)

Frontier evolution:

## 1. Length 0 Paths

[mo]

## 2. Length 1 Paths (inserted after 1 len1 path) *inserted at end of frontier in order shown*

[al,mo] [ls,mo] [ws,mo] [ch,mo]

## 3. Length 2 Paths (inserted after 4 len1 paths) *inserted at end of frontier in order shown*

[eif,al,mo] [sec,ls,mo] [fs,ws,mo] [fs,ch,mo]  
                          [sec,ws,mo] [ac,ch,mo]  
                          [nyse,ws,mo] [trp,ch,mo]  
                          [ac,ws,mo]

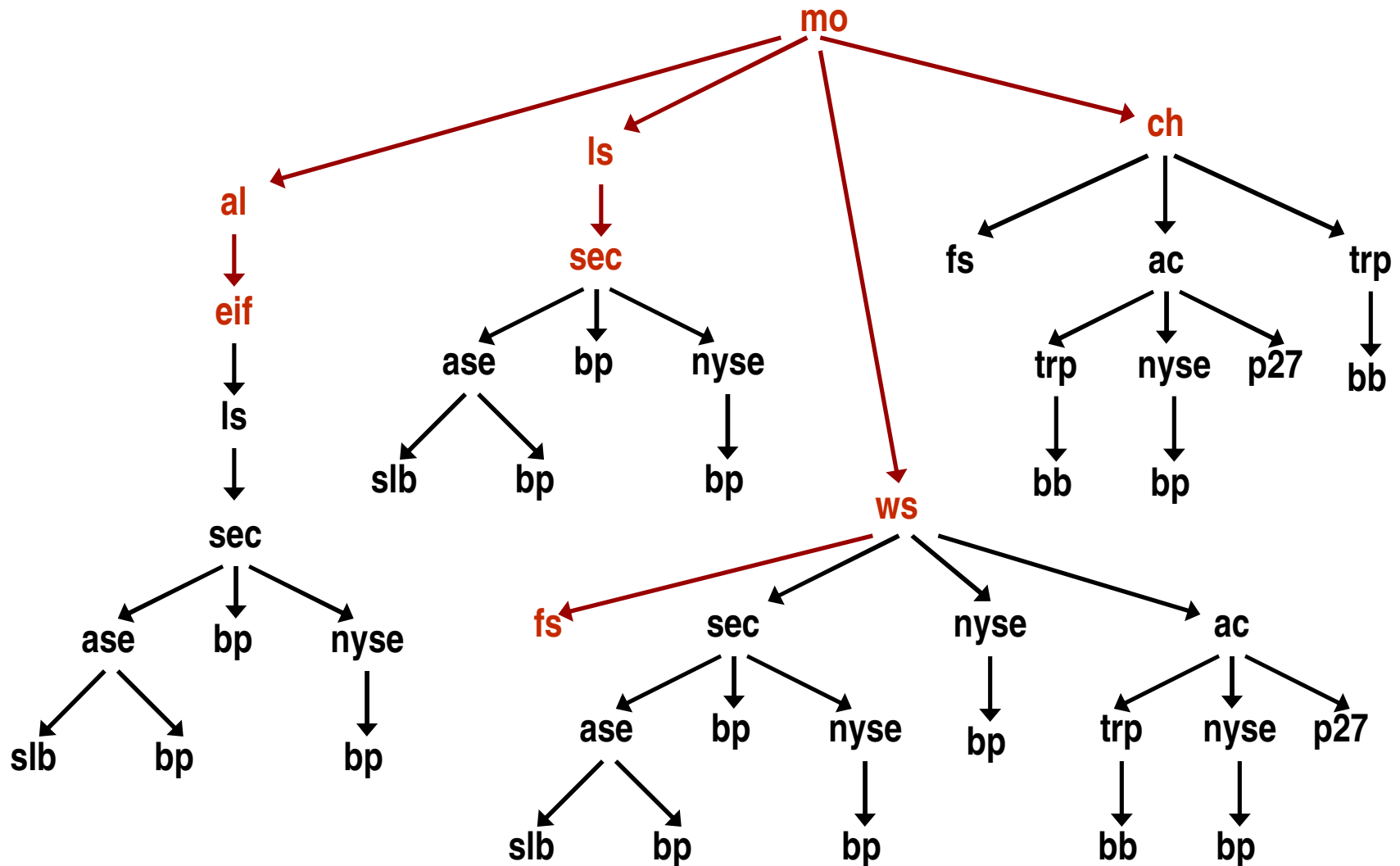
## 4. Length 3 Paths (inserted after 9 len2 paths) *inserted at end of frontier in order shown*

[ls,eif,al,mo] [ase,sec,ls,mo] (plus 14 more paths not added to frontier)  
                  [bp,sec,ls,mo]  
                  [nyse,sec,ls,mo]

On Third Step  
(3rd Path in the Length  
2 Frontier),  
the goal FS will be found.

Total Search steps:  
 $1+4+3 = 8$   
8 nodes expanded

# Paths Explored by BFS in Example



# Properties of BFS

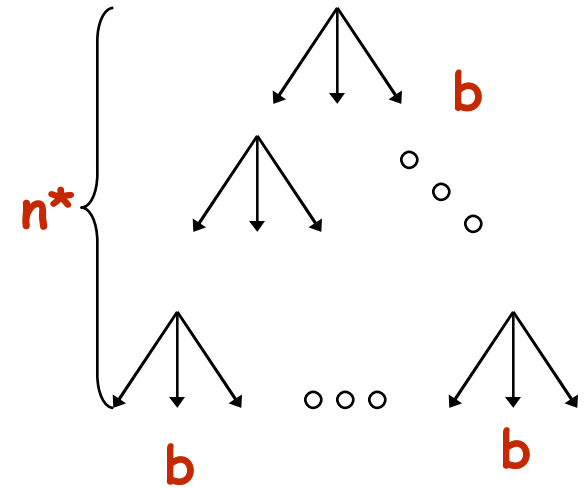
- All paths of length  $k$  occur on frontier after all length  $k-1$  paths
- No length  $k$  path is added to frontier until we have expanded all length  $k-1$  paths
- This last property ensures that we are guaranteed to find shortest path (if sol'n exists)
  - If we find a length  $k$  sol'n, since we've looked at all shorter paths, no shorter sol'n exists

# Properties of BFS: Time

## ■ How long can BFS take?

### (a) Finite graph, no cycles:

- If branching factor bounded by  $b$ , and the shortest sol'n has length  $n^*$ , then we'll explore  $O(b^{n^*})$  paths (length  $n$ )
- note: presence of cycles has no effect if a solution is present
- If no solution, will explore all paths:  $O(b|N|)$



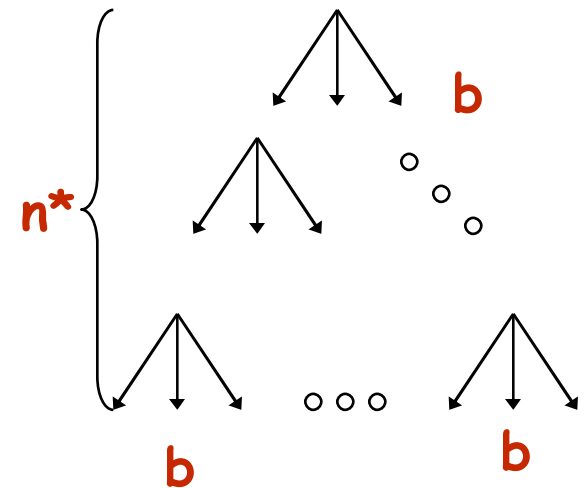
### (b) Finite graph with cycles:

- if sol'n: same as above
- if no sol'n: may not terminate unless we perform cycle checking or multiple path checking



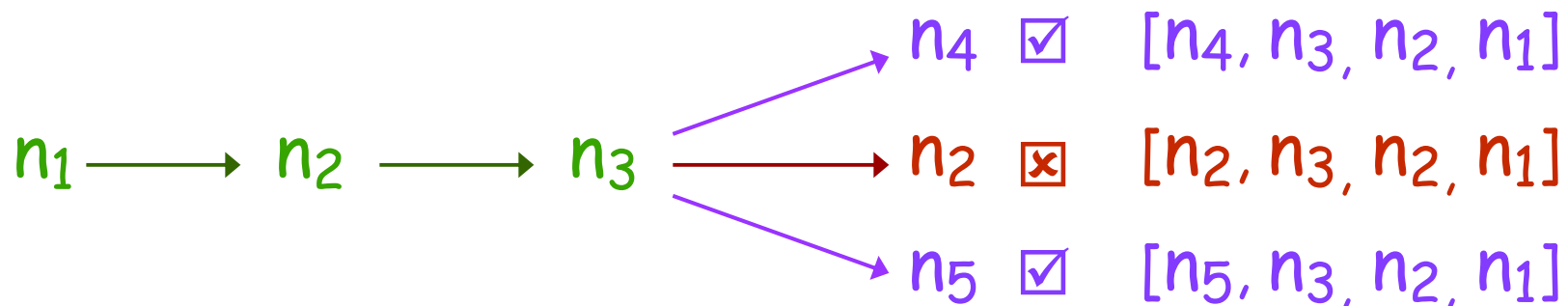
# Properties of BFS: Space

- How many paths on frontier at any one time?
  - If current path length  $m$ , then there are between  $b^{m-1}$  and  $b^m$  paths on frontier
  - If shortest path has length  $n^*$ , then guaranteed to have frontier of size  $O(b^{n^*})$
  - Luck with node ordering plays no role: BFS is systematic
- Space cost is the price you pay for optimality



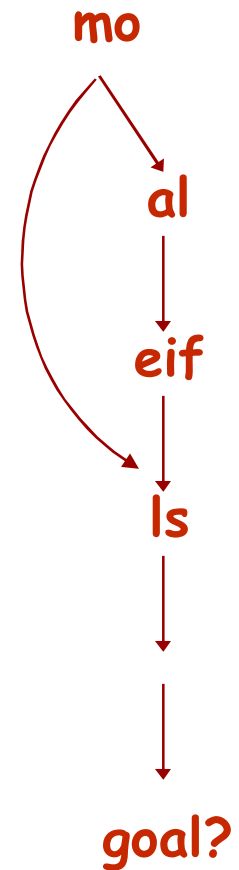
# Cycle Checking (see text for more)

- Cycles can hurt DFS: can prevent termination
- **Cycle checking** in DFS: requires a simple test
  - when you select a path from frontier and extend it with its neighbors, we only add a new path to the frontier if the neighbor is not already on the path
- Test requires linear time in length of path
  - some tricks can be used to reduce this
- Cannot affect existence of soln, rule out best soln



# Multiple Paths to Same Node

- In Manhattan example, with start = mo:
  - at depth 2, we have path  $P_1 = [ls, mo]$
  - at depth 4, we have path  $P_2 = [ls, eif, al, mo]$
- Why add  $P_2$  to the frontier?
  - If there is a path from  $ls$  to  $goal$ , then extension of  $P_1$  to the  $goal$  is shorter than the extension of  $P_2$  (each extension of  $P_2$  added to frontier is just wasted)
  - If there is no path from  $ls$  to  $goal$ , not adding  $P_2$  to frontier cannot hurt



# Multiple Path Checking in BFS

- Cycle checking can be applied to BFS too
  - saves some time (don't explore cyclic path)
  - ensures termination in cyclic graphs with no solution
- **Multiple path checking** is more general
  - Every time a (path to a) node is considered for addition to frontier, check list of *visited* nodes (those that have already been expanded).
  - If node is on list, do not add it to the frontier.
  - If node is not on list, add it to frontier and visited list.
- In BFS, need an extra argument: *VisitedList*
- MPC subsumes cycle checking

# Notes on MPC

- In BFS, we need an extra argument, *VisitedList*, to maintain the list of visited nodes
  - what would you initialize *VisitedList* with on first call?
- MPC *subsumes* cycle checking
  - a cycle is just one type of “multiple path” to same node
- Why doesn't MPC make sense for DFS?
- Exercise: Sketch out revised clauses defining:
  - DFS with cycle checking
  - BFS with MPC