CSC384: Lecture 4

- Last time
  - 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 multi-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

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?]
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 \( L : N \rightarrow \text{Labels} \)
  - denote properties of states (e.g., a value)
- An edge labeling is a function \( L : E \rightarrow \text{Labels} \)
  - denote properties of edges (e.g., a move cost)

Paths and Cycles

- A path in graph \( G = (N, E) \) is a sequence of nodes \( \langle n_1, n_2, \ldots, n_k \rangle \) s.t. each \( \langle n_i, n_{i+1} \rangle \in E \)
  - \( n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k \) is a path from \( n_1 \) to \( n_k \)
- A cycle is a path \( \langle n_1, n_2, \ldots, n_k \rangle \) 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
    - \( X = \text{find me any path to the office} \)
    - \( X = \text{P is the best path to goal G} \): this means we have some optimization criterion we need to satisfy
      - possible criteria: shortest (\# edges); least cost; etc.
      - Find me shortest/fastaest path to office
      - \( X = \text{P has quality q} \) (a satisfying 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
  • 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: is_goal(h).
Start Node: office

search([off]),
select(mr, RF = [h], not a goal, NF = [mr, h])
search([mr, h]),
select rf = [h], not a goal, NF = [lng, h]
search([lng]),
select rf = [h], not a goal, NF = [lng, h]
search(h),
select rf = [], is a goal, NF = [mr, h]
Return yes!

Search Tree for Modified Example

Numbering reflects order in which nodes are examined (isgoal?)

What if we added an edge from lng 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

The MBC Rep’n (No Costs)

<table>
<thead>
<tr>
<th>mo: Max Office</th>
<th>ch: City Hall</th>
<th>trp: T. Rowe Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>al: Alley</td>
<td>fa: Fark Street</td>
<td>dc: Diamond Consulting</td>
</tr>
<tr>
<td>eif: Ellis Island Ferry</td>
<td>wa: Wall Street</td>
<td>pi: Park 27</td>
</tr>
<tr>
<td>rp: Rock Park</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

- 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

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 [n3, n2, n1] and neighbors of nt specified by nb(n3, [n4, n5, n6]); it produces 3 new paths organized in a list:
    - [n4, n3, n2, n1], [n5, n3, n2, n1], [n6, n3, n2, n1]

DFS with Path Extraction

*search(F, [Node | Rest[]])* :-
- *select([Node | Rest[]], F, RemF), is_goal(Zero)*.
- *search(F, [Node | Rest[]])* :-
- *select([Node | Rest[]], F, RemF),
- *NB(Node, NBList)*,
- *extendpath(NBList, [Node | Rest[]], NewLaths), add_to_frontier(RemF, NewLaths, NewF)*.
- *search(NewF, [Lath]*)*.
- *select(Lath, [Lath | Rest|Laths], Rest|Laths)*.
- *add_to_frontier(RemF, Laths, NewF)* :-
- *append(Laths, RemF, NewF)*.

Search Tree: MBC Acyclic; Start *mo*

DFS with Path Extraction

*search(F, [Node | Rest[]])* :-
- *select([Node | Rest[]], F, RemF), is_goal(Zero)*.
- *search(F, [Node | Rest[]])* :-
- *select([Node | Rest[]], F, RemF),
- *NB(Node, NBList)*,
- *extendpath(NBList, [Node | Rest[]], NewLaths), add_to_frontier(RemF, NewLaths, NewF)*.
- *search(NewF, [Lath]*)*.
- *select(Lath, [Lath | Rest|Laths], Rest|Laths)*.
- *add_to_frontier(RemF, Laths, NewF)* :-
- *append(Laths, RemF, NewF)*.

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

Frontier evolution:
1. [mo]
2. [id, mo] [ls, mo] [ws, mo] [ch, mo] (0)
3. [eif, id, mo] (0)
4. [eif, ls, id, mo] (0)
5. [eif, ls, al, mo] (0)
6. [use, sec, mo]
   *bp, sec, mo* [nyse, sec, mo] (8)
7. [slb, use, mo]
   *bp, use, mo* [nyse, sec, mo] (0)
8. [bp, use, mo]
   *bp, use, mo* (0)
9. [bp, sec, mo] [nyse, sec, mo] (0)
10. [nyse, sec, mo] (0)

List of Search steps: 21 nodes expanded
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 solution 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 \(n\) (i.e., current node selected), then there are \(bn\) 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 \(b + 2b + 3b + \ldots + nb\) 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 solution: mo -> al -> elf -> ls, even though shorter solution mo -> is 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.

Frontier Growth in BFS

```prolog
select(Path, [Path|RestPaths], RestPaths).
add_1f(remF, Paths, NewF) :-
    append(remF, Paths, NewF).
```
Trace of BFS (with paths: mo to fs)

Frontier evolution:
1. Length 0 Paths
   [mo]
2. Length 1 Paths (inserted after 0) (len path) inserted at end of frontier in order shown
   [0,mo] [ls,me] [w,se,m] [ch,mo]
3. Length 2 Paths (inserted after 1) (len path) inserted at end of frontier in order shown
   [ris,al,mo] [se,la,mo] [fe,se,w,mo] [ge,se,ls,m] [ny,c,se,la,mo] [trp,al,ls,m] [or,se,la,mo]
4. Length 3 Paths (inserted after 2) (len path) inserted at end of frontier in order shown
   [ls,la,mo] [la,se,la,mo] [ls,la,se,la,mo] [ls,la,se,la,mo] (plus 14 more paths not added to frontier)

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^nK)

  (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 sol’n, rule out best sol’n

```latex
\begin{align*}
\text{n}_1 & \rightarrow \text{n}_2 \rightarrow \text{n}_3 \\
& \rightarrow [\text{n}_4, \text{n}_3, \text{n}_2, \text{n}_1] \\
& \rightarrow [\text{n}_2, \text{n}_3, \text{n}_2, \text{n}_1] \\
& \rightarrow [\text{n}_5, \text{n}_3, \text{n}_2, \text{n}_1]
\end{align*}
```
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 $P_2 = [ls, ef, al, mo]$
- 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