Trees and Search Strategies
and Algorithms --

Reference: Dr. Franz J. Kurfess
Computer Science Department
Cal Poly
Basic Search Strategies

– depth-first
– breadth-first

• exercise
  – apply depth-first to finding a path from this building to your favorite “feeding station” (McDonalds, Jason Deli, Pizza Hut)
    • is this task sufficiently specified
    • is success guaranteed
    • how long will it take
    • could you remember the path
    • how good is the solution
Motivation

- search strategies are important methods for many approaches to problem-solving
- the use of search requires an abstract formulation of the problem and the available steps to construct solutions
- search algorithms are the basis for many optimization and planning methods
Objectives

• formulate appropriate problems as search tasks
  – states, initial state, goal state, successor functions (operators), cost

• know the fundamental search strategies and algorithms
  • breadth-first, depth-first,

• evaluate the suitability of a search strategy for a problem
  – completeness, time & space complexity, optimality
Problems

– solution
  • path from the initial state to a goal state

– search cost
  • time and memory required to calculate a solution

– path cost
  • determines the expenses of the agent for executing the actions in a path
  • sum of the costs of the individual actions in a path

– total cost
  • sum of search cost and path cost
  • overall cost for finding a solution
Traveling Salesperson

- **states**
  - locations / cities
  - illegal states
    - each city may be visited only once
    - visited cities must be kept as state information

- **initial state**
  - starting point
  - no cities visited

- **successor function (operators)**
  - move from one location to another one

- **goal test**
  - all locations visited
  - agent at the initial location

- **path cost**: distance between locations
Searching for Solutions

- traversal of the search space
  - from the initial state to a goal state
  - legal sequence of actions as defined by successor function (operators)

- general procedure
  - check for goal state
  - expand the current state
    - determine the set of reachable states
    - return “failure” if the set is empty
  - select one from the set of reachable states
  - move to the selected state

- a search tree is generated
Search Terminology

• search tree
  – generated as the search space is traversed
    • the search space itself is not necessarily a tree, frequently it is a graph
    • the tree specifies possible paths through the search space
  – expansion of nodes
    • as states are explored, the corresponding nodes are expanded by applying the successor function
      – this generates a new set of (child) nodes
    • the fringe (frontier) is the set of nodes not yet visited
      – newly generated nodes are added to the fringe
  – search strategy
    • determines the selection of the next node to be expanded
    • can be achieved by ordering the nodes in the fringe
      – e.g. queue (FIFO), stack (LIFO), “best” node w.r.t. some measure (cost)
Example: Graph Search

- the graph describes the search (state) space
  - each node in the graph represents one state in the search space. e.g. a city to be visited in a routing or touring problem
- this graph has additional information
  - names and properties for the states (e.g. S, 3)
  - links between nodes, specified by the successor function
    - properties for links (distance, cost, name, ...)
Breadth First Search
Graph and Tree

- The tree is generated by traversing the graph.
- The same node in the graph may appear repeatedly in the tree.
- The arrangement of the tree depends on the traversal strategy (search method).
- The initial state becomes the root node of the tree.
- In the fully expanded tree, the goal states are the leaf nodes.
Greedy Search
A* Search
function TREE-SEARCH(problem, fringe) returns solution
    fringe := INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if EMPTY?(fringe) then return failure
        node := REMOVE-FIRST(fringe)
        if GOAL-TEST[problem] applied to STATE[node] succeeds
            then return SOLUTION(node)
        fringe := INSERT-ALL(EXPAND(node, problem), fringe)
    end loop

• generate the node from the initial state of the problem
• repeat
  – return failure if there are no more nodes in the fringe
  – examine the current node; if it’s a goal, return the solution
  – expand the current node, and add the new nodes to the fringe
function GENERAL-SEARCH(problem, QUEUING-FN) returns solution
nodes := MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))

loop do
    if nodes is empty then return failure
    node := REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds
        then return node
    nodes := QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end
Evaluation Criteria

- **completeness**
  - if there is a solution, will it be found

- **time complexity**
  - how long does it take to find the solution
  - does not include the time to perform actions

- **space complexity**
  - memory required for the search

- **optimality**
  - will the best solution be found

Main factors for complexity considerations:

- branching factor $b$, depth $d$ of the shallowest goal node, maximum path length $m$
Search Cost

• the *search cost* indicates how expensive it is to generate a solution
  – time complexity (e.g. number of nodes generated) is usually the main factor
  – sometimes space complexity (memory usage) is considered as well

• *path cost* indicates how expensive it is to execute the solution found in the search
  – distinct from the search cost, but often related

• *total cost* is the sum of search and path costs
Breadth-First

- all the nodes reachable from the current node are explored first
  - achieved by the TREE-SEARCH method by appending newly generated nodes at the end of the search queue

```python
function BREADTH-FIRST-SEARCH(problem) returns solution

    return TREE-SEARCH(problem, FIFO-QUEUE())
```

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>(b^{d+1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space Complexity</td>
<td>(b^{d+1})</td>
</tr>
<tr>
<td>Completeness</td>
<td>yes (for finite (b))</td>
</tr>
<tr>
<td>Optimality</td>
<td>yes (for non-negative path)</td>
</tr>
</tbody>
</table>

- \(b\) branching factor
- \(d\) depth of the tree
Breadth-First Snapshot 1

Fringe: [] + [2,3]
Breadth-First Snapshot 2

Fringe: \([3] + [4,5]\)
Breadth-First Snapshot 3

Fringe: [4,5] + [6,7]
Breadth-First Snapshot 4

Fringe: [5,6,7] + [8,9]
Breadth-First Snapshot 5

Fringe: [6,7,8,9] + [10,11]
Breadth-First Snapshot 6

Fringe: [7,8,9,10,11] + [12,13]
Breadth-First Snapshot 7

Fringe: [8,9,10,11,12,13] + [14,15]
Breadth-First Snapshot 8

Fringe: [9,10,11,12,13,14,15] + [16,17]
Breadth-First Snapshot 9

Fringe: [10, 11, 12, 13, 14, 15, 16, 17] + [18, 19]
Breadth-First Snapshot 10

Fringe: [11,12,13,14,15,16,17,18,19] + [20,21]
Breadth-First Snapshot 11

Fringe: [12, 13, 14, 15, 16, 17, 18, 19, 20, 21] + [22, 23]
Breadth-First Snapshot 12

Fringe: [13,14,15,16,17,18,19,20,21] + [22,23]

Note: The goal node is “visible” here, but we cannot perform the goal test yet.
Breadth-First Snapshot 13

Fringe: [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25] + [26, 27]
Breadth-First Snapshot 14

Fringe: [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27] + [28, 29]
Breadth-First Snapshot 15

Fringe: [15,16,17,18,19,20,21,22,23,24,25,26,27,28,29] + [30,31]
Breadth-First Snapshot 16

Fringe: [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31]
Breadth-First Snapshot 17

Fringe: [18,19,20,21,22,23,24,25,26,27,28,29,30,31]
Breadth-First Snapshot 18

Fringe: [19,20,21,22,23,24,25,26,27,28,29,30,31]
Breadth-First Snapshot 19

Fringe: [20,21,22,23,24,25,26,27,28,29,30,31]
Breadth-First Snapshot 20

Fringe: [21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
Breadth-First Snapshot 21

Fringe: [22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
Breadth-First Snapshot 22

Fringe: [23, 24, 25, 26, 27, 28, 29, 30, 31]
Breadth-First Snapshot 23

Fringe: [24,25,26,27,28,29,30,31]
Breadth-First Snapshot 24

Initial
Visited
Fringe
Current
Visible
Goal

Note: The goal test is positive for this node, and a solution is found in 24 steps.

Fringe: [25, 26, 27, 28, 29, 30, 31]
Depth-First

function DEPTH-FIRST-SEARCH(problem) returns solution

return TREE-SEARCH(problem, LIFO-QUEUE())

• continues exploring newly generated nodes
  – achieved by the TREE-SEARCH method by appending newly generated nodes at the beginning of the search queue
  • utilizes a Last-In, First-Out (LIFO) queue, or stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time Complexity</strong></td>
<td>$b^m$</td>
</tr>
<tr>
<td><strong>Space Complexity</strong></td>
<td>$b \times m$</td>
</tr>
<tr>
<td><strong>Completeness</strong></td>
<td>no</td>
</tr>
<tr>
<td><strong>Optimality</strong></td>
<td>no</td>
</tr>
</tbody>
</table>

$b$ branching factor
$m$ maximum path length
Depth-First Snapshot

Fringe: [3] + [22,23]
Depth-First vs. Breadth-First

• depth-first goes off into one branch until it reaches a leaf node
  – not good if the goal is on another branch
  – neither complete nor optimal
  – uses much less space than breadth-first
    • much fewer visited nodes to keep track of
    • smaller fringe
• breadth-first is more careful by checking all alternatives
  – complete and optimal
  – very memory-intensive