COMP9414: Artificial Intelligence

Lecture 2a: Problem Solving

Wayne Wobcke

e-mail:w.wobcke@unsw.edu.au

This Lecture

- Search as a "weak method" of problem solving with wide applicability
- Uninformed search methods (use no problem-specific information)
- Informed search methods (use heuristics to improve efficiency)

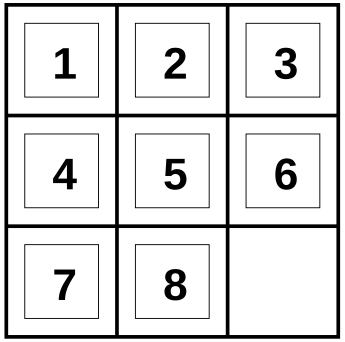
Motivating Example

- You are in Romania on holiday, in Arad, and need to get to Bucharest
- What more information do you need to solve this problem?
- Once you have this information, how do you solve the problem?
- How do you know your solution is any good? What extra information would you need in order to evaluate the quality of your solution?

State Space Search Problems

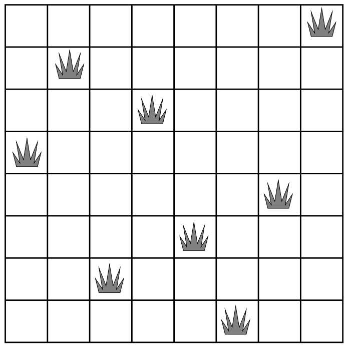
- State space – set of all states reachable from initial state(s) by any action sequence
- Initial state(s) – element(s) of the state space
- Transitions
  - Operators – set of possible actions at agent's disposal; describe state reached after performing action in current state, or
  - Successor function – $s(x)$= set of states reachable from state x by performing a single action
- Goal state(s) – element(s) of the state space
- Path cost – cost of a sequence of transitions used to evaluate solutions (apply to optimization problems)

Example Problem – 8-Puzzle



States: location of eight tiles plus location of blank
Operators: move blank left, right, up, down
Goal state: state with tiles arranged in sequence
Path cost: each step is of cost 1

Example Problem – N-Queens



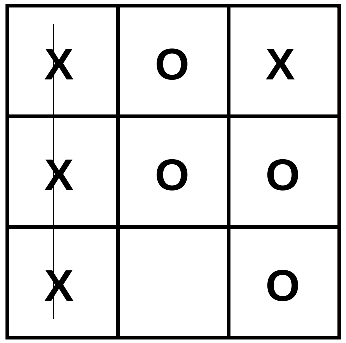
States: 0 to N queens arranged on chess board
Operators: place queen on empty square
Goal state: N queens on chess board, none attacked
Path cost: zero

Real World Problems

- Route finding – robot navigation, airline travel planning, computer/phone networks
- Travelling salesman problem – planning movement of automatic circuit board drills
- VLSI layout – design silicon chips
- Assembly sequencing – scheduling assembly of complex objects, manufacturing process control
- Mixed/constrained problems – courier delivery, product distribution, fault service and repair

These are optimization problems but mathematical (operations research) techniques are not always effective.

Problem Representation – Tic-Tac-Toe



States: arrangement of Os and Xs on 3x3 grid
Operators: place X (O) in empty square
Goal state: three Xs (Os) in a row
Path cost: zero

Tic-Tac-Toe – First Attempt

1	2	3
4	5	6
7	8	9

Board: 0=blank; 1=X; 2=O
Idea: Use move table with $3^9 = 19683$ elements
Algorithm: Consider board to be a ternary number; convert to decimal; access move table; update board
- Fast; lots of memory; laborious; not extensible

Tic-Tac-Toe – Second Attempt

1	2	3
4	5	6
7	8	9

Board: 2=blank; 3=X; 5=O
Algorithm: Separate strategy for each move.
Goal test (if row gives win on next move): calculate product of values
X: test product = 18 ($3 \times 3 \times 2$); O: test product = 50 ($5 \times 5 \times 2$)
- Not as fast as 1; much less memory; easier to understand and comprehend; strategy determined in advance; not extensible

Tic-Tac-Toe – Third Attempt

8	3	4
1	5	9
6	7	2

Board is a magic square!
Algorithm: As in attempt 2 but to check for win – keep track of player's "squares".
If difference of 15 and sum of two squares is ≤ 0 or > 9 two squares are not collinear. Otherwise, if square equal to difference is blank, move there.
- What does this tell you about the way humans solve problems vs computers?

Tic-Tac-Toe – Fourth Attempt

	?	

Board: list of board positions arising from next move; estimate of likelihood of position leading to a win
Algorithm: look at position arising from each move; choose "best" one
- Slower; can handle large variety of problems

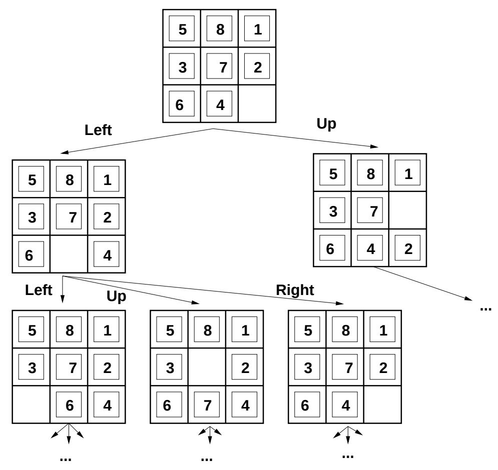
Back to Motivating Example

- Notice assumptions built in to problem formulation (level of abstraction)
- Note that while people can "look" at the map to see a solution, the computer must construct the map by exploration
  - Where can I go from Arad?
  - Sibiu, Timisoara, Zerind
  - Where can I go from Sibiu?
- The order of questioning defines the search strategy
- Problem formulation assumptions critically affect the quality of the solution to the original problem

Explicit State Spaces

- View state space search in terms of finding a path through a graph
- Graph $G = (V, E)$ – V: vertices; E: edges
- Edges may have associated cost; path cost = sum edge costs in path
- Path from vertex s to g – sequence of vertices $s = n_0, n_1, \ldots, n_k = g$ such that there is an edge from n_i to n_{i+1}
- State space graph – node represents state; edge represents change from one state to another due to action; costs may be associated with vertices and edges (hence paths)
- Forward (backward) branching factor – max #out-(in-)going arcs from (to) node

State Space – 8-Puzzle



Complications

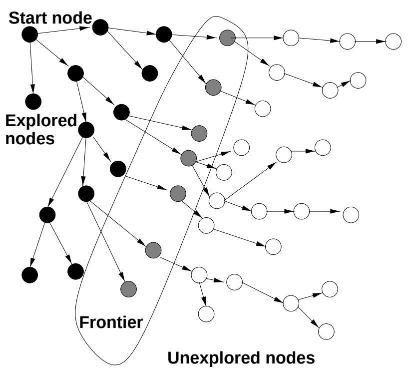
- Single-state – agent starts in known world state and knows which unique state it will be in after a given action
- Multiple-state – limited access to world state means agent is unsure of world state but may be able to narrow it down to a set of states
- Contingency problem – if agent does not know full effects of actions (or there are other things going on) it may have to sense during execution (changing the search space dynamically)
- Exploration problem – no knowledge of effects of actions (or state), so agent must experiment

Search methods are capable of tackling single-state problems and multiple-state problems at the cost of additional complexity

Uninformed (Blind) Search Algorithms

- Breadth-First Search
- Uniform Cost Search
- Depth-First Search
- Depth-Limited Search
- Iterative Deepening Search
- Bidirectional Search

General Search Space (not State Space)



Search strategy – way in which frontier expands

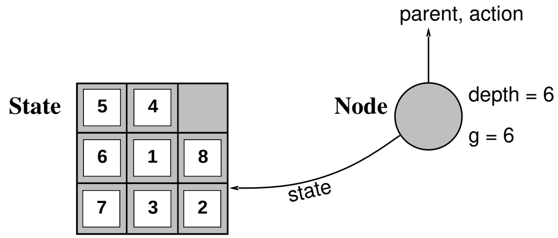
General Search Procedure

function GeneralSearch(problem, strategy) **returns** a solution or failure
initialize search graph using the initial state of problem
  loop
    if there are no candidates for expansion **then return** failure
    choose a frontier node for expansion according to strategy
    if the node contains a goal state **then return** solution
    else expand the node and add the resulting nodes to the search graph
end

Note: Only test whether at goal state when expanding node, not when adding nodes to the search graph (except for breadth-first search!)

State Space vs Search Space

- A state is part of the formulation of the search problem
- A node is a data structure used in a search graph/tree, and includes:
  - parent, operator, depth, path cost $g(x)$
- States do not have parents, children, depth, or path cost!



Two different nodes can have the same state

Evaluating Search Algorithms

- Completeness: strategy guaranteed to find a solution when one exists?
- Time complexity: how long to find a solution?
- Space complexity: memory required during search?
- Optimality: when several solutions exist, does it find the "best"?

Note: States are constructed during search, not computed in advance, so efficiently computing successor states is critical!

Analysis of Algorithms – Big-O

- $T(n)$ is $O(f(n))$ means that there is some n_0 and k such that

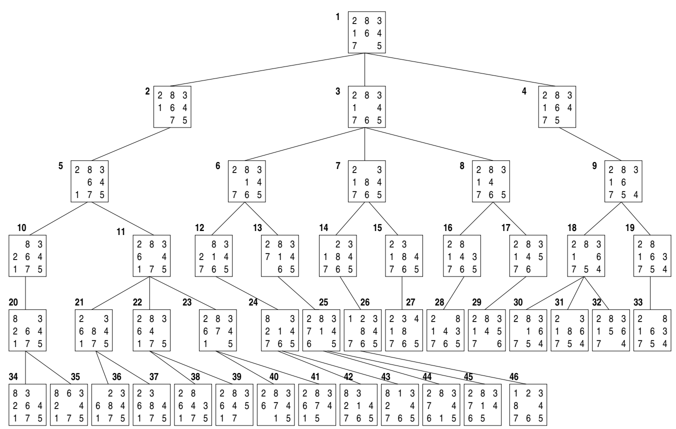
  $$T(n) \leq kf(n) \text{ for every problem of size } n \geq n_0$$

- Independent of implementation, compiler, fixed overheads, ...
- $O()$ abstracts over constant factors
- Examples
  - $O(n)$ algorithm is better than an $O(n^2)$ algorithm (in the long run)
  - $T(100 \cdot n + 1000)$ is better than $\mathrm{T}(n^2 + 1)$ for $n > 110$
  - Polynomial $O(n^k)$ much better than exponential $O(2^n)$
- $O()$ notation is a compromise between precision and ease of analysis

Breadth-First Search

- **Idea:** Expand root node, then expand all children of root, then expand their children, ...
- All nodes at depth d are expanded before nodes at $d+1$
- Can be implemented by using a queue to store frontier nodes
- Breadth-first search finds shallowest goal state
- Stop when node with goal state is generated
- Include check that generated state has not already been explored
  - Needs a new data structure for set of explored states

Breadth-First Search



Breadth-First Search – Analysis

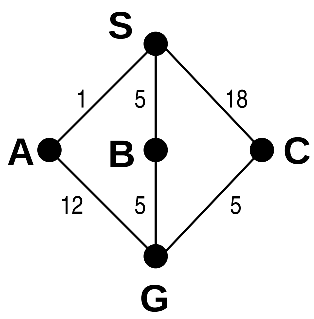
- Complete
- Optimal – provided path cost is nondecreasing function of the depth of the node
- Maximum number of nodes generated: $b + b^2 + b^3 + \ldots + b^d$ (where b = forward branching factor; d = path length to solution)
- Time and space requirements are the same $O(b^d)$

Uniform Cost Search

- Also known as Lowest-Cost-First search
- Shallowest goal state may not be the least-cost solution
- **Idea:** Expand lowest cost (measured by path cost $g(n)$) node
- Order nodes in the frontier in increasing order of path cost
- Breadth-first search $\approx$ uniform cost search where $g(n) = depth(n)$ (except breadth-first search stops when goal state generated)
- Include check that generated state has not already been explored
- Include test to ensure frontier contains only one node for any state – for path with lowest cost

Uniform Cost Search

Uniform cost search is optimal only if it stops when goal node is expanded – not when goal node is generated



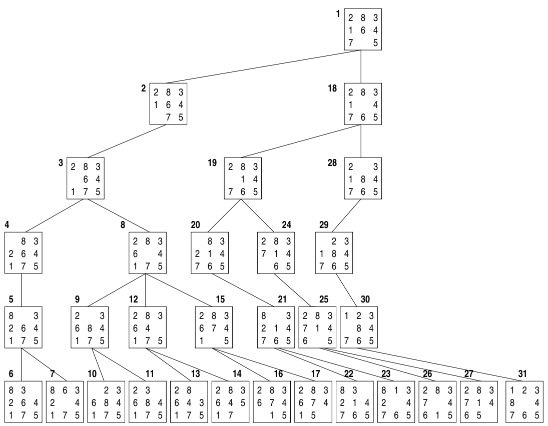
Uniform Cost Search – Analysis

- Complete
- Optimal – provided path cost does not decrease along path (i.e. $g(successor(n)) \geq g(n)$ for all n)
- Reasonable assumption when path cost is cost of applying operators along the path
- Performs like breadth-first search when $g(n) = depth(n)$
- If there are paths with negative cost, need exhaustive search

Depth-First Search

- **Idea:** Always expand node at deepest level of tree and when search hits a dead-end return back to expand nodes at a shallower level
- Can be implemented using a stack of explored + frontier nodes
- At any point depth-first search stores single path from root to leaf together with any remaining unexpanded siblings of nodes along path
- Stop when node with goal state is expanded
- Include check that generated state has not already been explored along a path – cycle checking

Depth-First Search



Depth-First Search

```
def search(self):
    """Returns (next) path from the start node to a goal node.
    Returns None if no path exists.
    """
    while not self.empty_frontier():
        path = self.frontier.pop()
        self.num_expanded += 1
        if self.problem.is_goal(path.end()):  # solution found
            self.solution = path    # store solution
            return path
        else:
            neighs = self.problem.neighbors(path.end())
            for arc in reversed(neighs):
                self.add_to_frontier(Path(path,arc))
    # No more solutions
```


Depth-First Search – Analysis

- Storage: $O(bm)$ nodes (where m = maximum depth of search tree)
- Time: $O(b^m)$
- In cases where problem has many solutions, depth-first search may outperform breadth-first search because there is a good chance it will find a solution after exploring only a small part of the space
- However, depth-first search may get stuck following a deep or infinite path even when a solution exists at a relatively shallow level
- Therefore, depth-first search is not complete and not optimal
- Avoid depth-first search for problems with deep or infinite paths

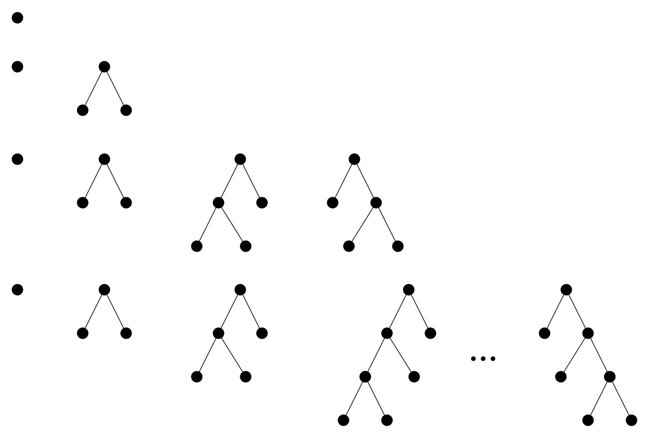
Depth-Limited Search

- **Idea:** Impose bound on depth of a path
- In some problems you may know that a solution should be found within a certain cost (e.g. a certain number of moves) and therefore there is no need to search paths beyond this point for a solution
- Analysis
  - Complete but not optimal (may not find shortest solution)
  - However, if the depth limit chosen is too small a solution may not be found and depth-limited search is incomplete in this case
  - Time and space complexity similar to depth-first search (but relative to depth limit rather than maximum depth)

Iterative Deepening Search

- It can be very difficult to decide upon a depth limit for search
- The maximum path cost between any two nodes is known as the diameter of the state space
- This would be a good candidate for a depth limit but it may be difficult to determine in advance
- **Idea:** Try all possible depth limits in turn
- Combines benefits of depth-first and breadth-first search

Iterative Deepening Search



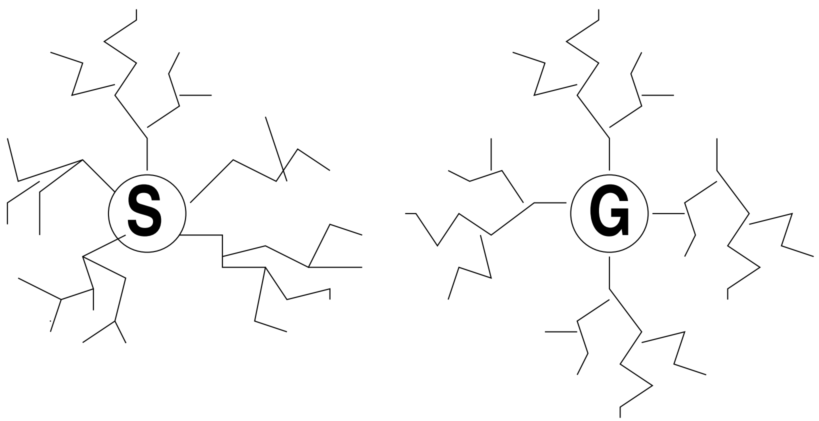
Iterative Deepening Search – Analysis

- Optimal; Complete; Space – $O(bd)$
- Some states are expanded multiple times: Isn't this wasteful?
  - Number of expansions to depth $d = 1 + b + b^2 + b^3 + \ldots + b^d$
  - Therefore, for iterative deepening, total expansions = $(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d$
  - The higher the branching factor, the lower the overhead (even for $b = 2$, search takes about twice as long)
  - Hence time complexity still $O(b^d)$
- Can double depth limit at each iteration – overhead $O(d\log d)$
- In general, iterative deepening is the preferred search strategy for a large search space where depth of solution is not known

Bidirectional Search

- **Idea:** Search forward from initial state and backward from goal state at the same time until the two meet
- To search backwards we need to generate predecessors of states (this is not always possible or easy)
- If operators reversible, successor sets and predecessor sets are the same
- If there are many goal states, maybe multi-state search would work (but not in chess)
- Need to check whether a node occurs in both searches – can be inefficient
- Which is the best search strategy for each half?

Bidirectional Search



Bidirectional Search – Analysis

- If solution exists at depth d then bidirectional search requires time $O(2b^{\frac{d}{2}}) = O(b^{\frac{d}{2}})$ (assuming constant time checking of intersection)
- To check for intersection must have all states from one of the searches in memory, therefore space complexity is $O(b^{\frac{d}{2}})$

Summary – Blind Search

Criterion	Breadth First	Uniform Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional
Time	b^d	b^d	b^m	b^l	b^d	$b^{\frac{d}{2}}$
Space	b^d	b^d	bm	bl	bd	$b^{\frac{d}{2}}$
Optimal	Yes	Yes	No	No	Yes	Yes
Complete	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b – branching factor
d – depth of shallowest solution
m – maximum depth of tree
l – depth limit

