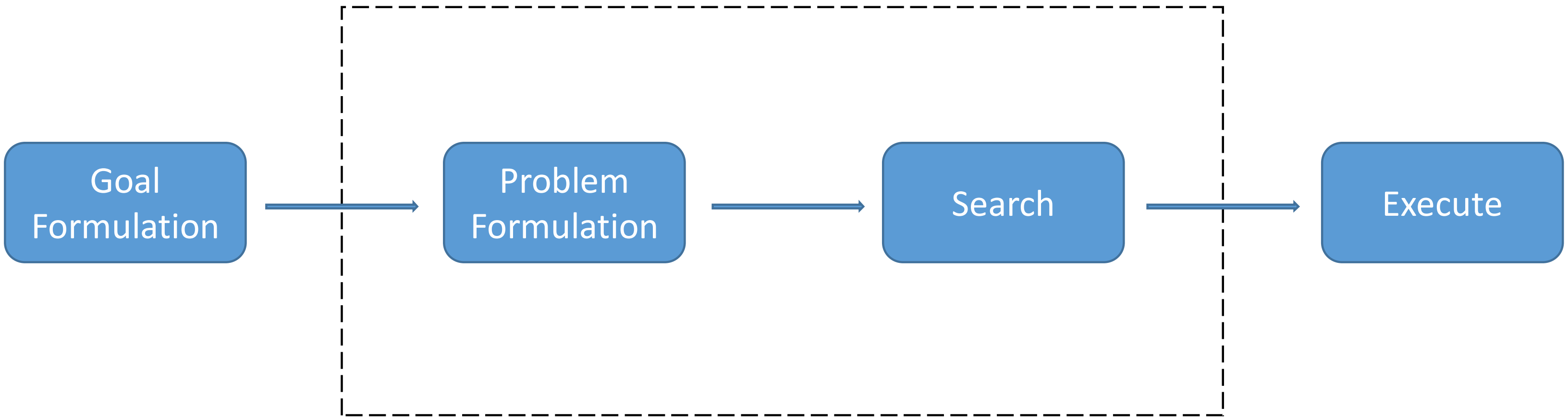
CS 472 Recitation

Week 3

Problem Solving



Problem Formulation

- States: All the cases, legal and illegal
- Initial State: It should be one in the States
- Actions/Transition Model: $State_i$ -> $State_{i+1}$
- Goal Test: Check whether achieve the goal or not
- Path Cost: If no specification, define it by yourself

Example- Sudoku Solver

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Each of the digits `1-9` must appear exactly once in

1. Each row
2. Each column
3. Each of the 9 `3x3` sub-boxes of the grid

Example- Sudoku Solver

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

States: [Empty,1-9]^81 = 10^81
Initial state: The given initial state
Actions, Transition model: Reset an empty spot to [1-9]
Goal test: If all spots are filled with digits and legal
Path cost: Number of actions

Search

Tree Search

```
function TREE-SEARCH(problem) return a solution or failure
  fringe ← INSERT(MAKE-NODE(problem.INITIAL-STATE))
  loop do
      if EMPTY?(fringe) then return failure
      node ← POP(fringe)
      if problem.GOAL-TEST(node.STATE)
              then return SOLUTION(node)
      for each action in problem.ACTIONS(node.STATE) do
              child ← Child-Node(problem, node, action)
              fringe ← INSERT(child, fringe)
```

Graph Search

```
function GRAPH-SEARCH(problem) return a solution or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(problem.INITIAL-STATE))
  loop do
      if EMPTY?(fringe) then return failure
      node ← POP(fringe)
      if problem.GOAL-TEST(node.STATE)
              then return SOLUTION(node)
      add node.STATE to closed
      for each action in problem.ACTIONS(node.STATE) do
              child ← Child-Node(problem, node, action)
              if child.STATE is not in closed or fringe then
                      fringe ← INSERT(child, fringe)
```

Problem: Repeated state

Search - Performance measure

- **Completeness:** Does it always find a solution if one exists?
- **Optimality:** Does it always find the least-cost solution?
- **Time Complexity:** Number of nodes generated
- **Space Complexity :** Number of nodes and edges stored in memory during search

  - ♦ d: *depth* (number of actions in the optimal solution)
  - ♦ m: *maximum number of actions* in any path
  - ♦ b: *branching factor* (number of successors of a node).

BF search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Criteria	Performance
Completeness	Yes (if branching factor b is finite)
Optimality	Unit cost, yes General cost, no
Time Complexity	Tree search: $O(b^d)$ Graph search: size of state space
Space Complexity	Same as time complexity

Uniform-Cost Search

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

function BEST-FIRST-SEARCH(*problem*, f) **returns** a solution node or *failure*
  node ← NODE(STATE=*problem*.INITIAL)
  frontier ← a priority queue ordered by f, with *node* as an element // states on the frontier
  reached ← a lookup table, with one entry with key *problem*.INITIAL and value *node* // states
  while not IS-EMPTY(*frontier*) **do** // that have been reached
    node ← POP(*frontier*)
    if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
    for each *child* **in** EXPAND(*problem*, *node*) **do**
      s ← *child*.STATE
      if s is not in *reached* **or** *child*.PATH-COST < *reached*[s].PATH-COST **then**
        reached[s] ← *child*
        add *child* to *frontier*
  return *failure*

function EXPAND(*problem*, *node*) **yields** nodes
  s ← *node*.STATE
  for each *action* **in** *problem*.ACTIONS(s) **do**
    s' ← *problem*.RESULT(s, *action*)
    cost ← *node*.PATH-COST + *problem*.ACTION-COST(s, *action*, s')
    yield NODE(STATE=s', PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

Criteria	Performance
Completeness	Yes, if step cost is positive (negative cost might cause infinite loops)
Optimality	Yes, if complete
Time Complexity	Tree search: $O(b^d)$ Graph search: size of state space
Space Complexity	Same as time complexity

DF Search

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Criteria	Performance
Completeness	No, infinite state space Graph: complete in finite state space Tree: complete in finite depth
Optimality	No, it gives the first solution which might not the optimal one
Time Complexity	Graph: size of state space Tree: $O(b^m)$ (m depth of the tree)
Space Complexity	Graph: size of state space Tree: $O(bm)$

Depth-limited Search

```
function DEPTH-LIMITED-SEARCH(problem,limit) return a solution or
    failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE),
    problem,limit)

function RECURSIVE-DLS(node, problem, limit) return a solution or
    failure/cutoff
    cutoff_occurred? ← false
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    else if DEPTH[node] == limit then return cutoff
    else for each action in problem.ACTIONS(node.STATE) do
        child ← Child-Node(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit)
        if result == cutoff  then cutoff_occurred? ←  true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

Criteria	Performance
Completeness	No, if depth limit l < solution depth d
Optimality	No
Time Complexity	$O(b^l)$
Space Complexity	$O(bl)$

Iterative Deepening search

```
function ITERATIVE_DEEPENING_SEARCH(problem)
  return a solution or failure

  for depth ← 0 to ∞ do
      result ← DEPTH-LIMITED_SEARCH(problem,
  depth)
      if result ≠ cuttoff  then return result
```

Criteria	Performance
Completeness	Yes
Optimality	Yes, if step cost is 1 (each step has the same cost)
Time Complexity	$O(b^l)$
Space Complexity	$O(bl)$

Best-first Search

Best First Search →

- Uniform-Cost Search: $f(n) = g(n)$
- Greedy Best First Search: $f(n) = h(n)$
- A* Search: $f(n) = g(n) + h(n)$

$g(n)$ = the actual cost from the initial state to the current state

$h(n) =$ *estimated* cost of the cheapest path from the state at node n to a goal state

Greedy Search

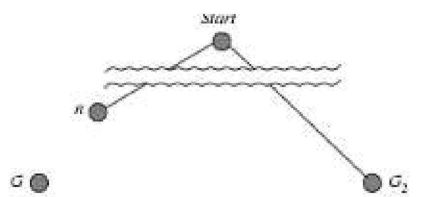
Criteria	Performance
Completeness	Complete with finite state space graph search, otherwise, not complete
Optimality	No
Time Complexity	Tree search $O(b^m)$ Graph search: size of state space
Space Complexity	Keep all nodes in memory Same as time complexity

A* Search

Criteria	Performance
Completeness	Complete with finite state space graph search, otherwise, not complete
Optimality	Yes, if the heuristic function is admissible
Time Complexity	Tree search $O(b^m)$ Graph search: size of state space
Space Complexity	Keep all nodes in memory Same as time complexity

Heuristic function $h(n)$

Admissible: $h(n) \le h^*(n)$



$f(G_2)$	$= g(G_2)$	since $h(G_2)=0$
	$> g(G)=g(n)+h^*(n)$	since G_2 is suboptimal
	$>= f(n)$	since h is admissible

- G_2 is suboptimal, it is a goal state but with larger cost than the optimal goal state G
- n is an unexpanded node in the frontier and it is on the path to G

Consistent: $h(n) \le c(n, n') + h(n')$

$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n, n') + h(n') \\
&\ge g(n) + h(n) \\
&\ge f(n)
\end{aligned}
$$

- n' is one child on n
- f(n) is non-decreasing along any path

Admissible Consistent

$$
\begin{aligned}
f(n) \le f(G) &= g(n) + h^*(n) \\
g(n) + h(n) &\le g(n) + h^*(n) \\
h(n) &\le h^*(n)
\end{aligned}
$$