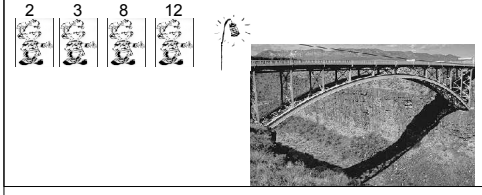
Uninformed Search

Louis Oliphant
oliphant@cs.wisc.edu
CS540 section 2

Example: Crossing the Bridge

- Objective: All people across the bridge in shortest time
- Problem:
  - At most 2 can travel across the bridge at a time
  - Must have lantern when travelling
  - Travel the time of the slowest person

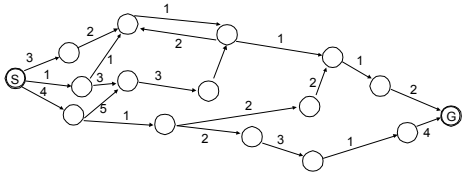


Defining the Environment

- State space : all valid configurations
- Initial (Start) and Goal states (can have more than one)
- Successor function : *successors*(**s**) are all states that can be reached from **s** by some legal action
- Cost : the cost to take an action from a state

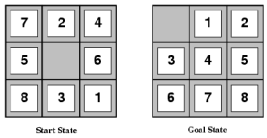
This information can be represented as a weighted, directed graph

Graph Representation of Environment



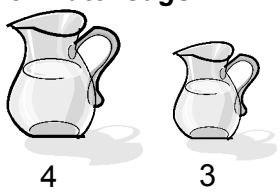
Objective : find path from Start to Goal
- Any path
- Shortest Path
- Least Cost Path

Example: 8-Puzzle



- States = configurations
- Sucessor function = up to 4 kinds of movement
- Cost = 1 for each move

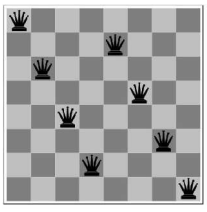
Example: Water Jugs



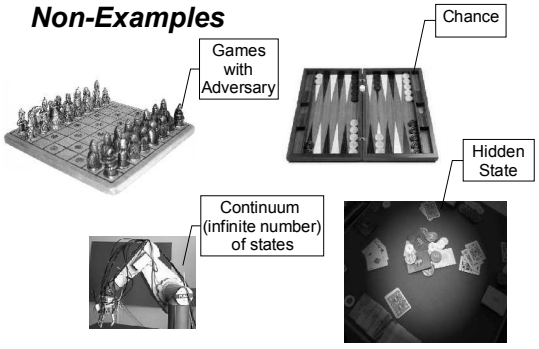
- Goal: get 2 gallons in 4 gallon jug
- Successor function: fill up, empty, or pour from one jug to the other

8-queens

- State: complete configuration vs. column-by-column



Non-Examples



Defining the Search

- Solution : A series of actions that will get you from the initial state to the goal state
- Node: A data structure containing
  - state of environment
  - parent node
  - action taken to reach this node
  - book keeping information
- Fringe: set of nodes that have been generated but not yet expanded
- Goal Test: function that returns true if node contains a goal state
- Expand: function that returns all children of a node

General Search Algorithm

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

General Search Algorithm

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

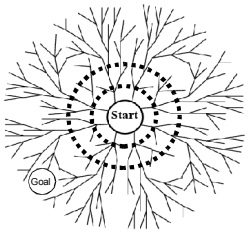
Interesting part are the possible methods of inserting into the Fringe

Methods of Inserting into Fringe

- Take shallowest node: Breadth First Search (BFS)
- Take least cost node: Uniform Cost Search(UCS)
- Take deepest node: Depth First Search (DFS)
- Take best node: need a scoring function and heuristics
- Semi-Randomly select node: Genetic Algorithms and Simulated Annealing
- And others like IDS, Bidirectional search

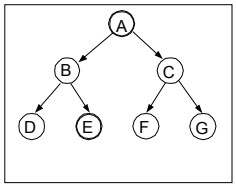
Breadth-first search (BFS)

- Expand the shallowest node first
  - Examine states **one** step away from the initial states
  - Examine states **two** steps away from the initial states
  - and so on…
- ripple



BFS in action

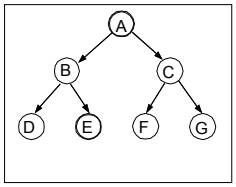
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node

BFS in action

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

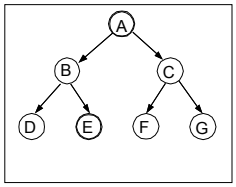


fringe	node

Fringe is a FIFO Queue

BFS in action

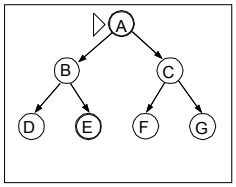
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	

BFS in action

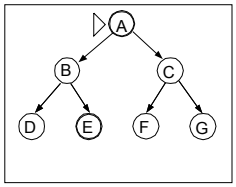
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
	A

BFS in action

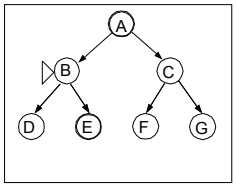
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A

BFS in action

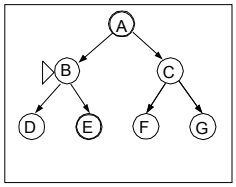
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
C	B

BFS in action

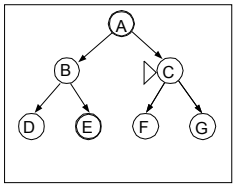
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B

BFS in action

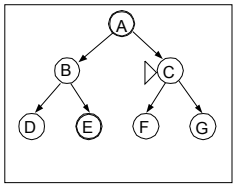
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
ED	C

BFS in action

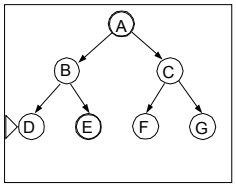
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
GFED	C

BFS in action

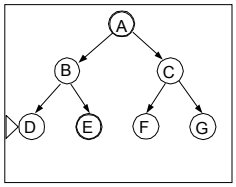
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
GFED	C
GFE	D

BFS in action

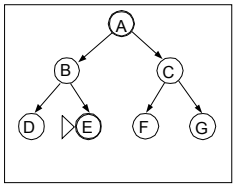
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
GFED	C
GFE	D

BFS in action

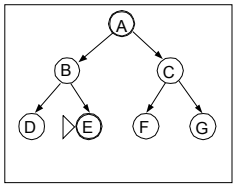
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
GFED	C
GFE	D
GF	E

BFS in action

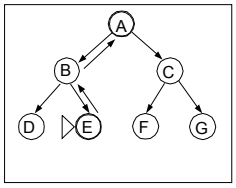
fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))



fringe	node
A	
CB	A
EDC	B
GFED	C
GFE	D
GF	E<-Goal!

BFS in action

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

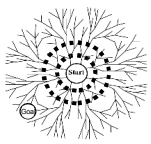


fringe	node
A	
CB	A
EDC	B
GFED	C
GFE	D
GF	E<-Goal!

-Follow back pointers to recover solution (path to goal)
-Every child keeps track of its parent

Performance of BFS

- Assume:
  - the graph may be infinite.
  - Goal(s) exists and is only finite steps away.
- Will BFS find at least one goal?
- Will BFS find the least cost goal?
- Time complexity?
  - # states generated
  - Goal ***d*** edges away
  - Branching factor ***b***
- Space complexity?
  - # states stored



Performance of BFS

Four measures of search algorithms:

- Complete: yes, if branching factor finite.
- Optimal: yes if edge costs identical.
- Time complexity (worst case): goal is the last node at depth ***d***.
  - Have to generate all nodes at radius ***d+1***.
  - $b + b^2 + \ldots + b^d + b^{d+1} \sim O(b^{d+1})$
- Space complexity (bad)
  - Back pointers for all generated nodes $O(b^{d+1})$
  - The queue / fringe (smaller, but still $O(b^{d+1})$)

Performance of BFS

Four measures of search algorithms: Uniform Cost

- Complete: yes, if branching factor finite.
- Optimal: yes if edge costs identical.
- Time complexity (worst case): goal is the last node at depth ***d***.
  - Have to generate all nodes at radius ***d+1***.
  - $b + b^2 + \ldots + b^d + b^{d+1} \sim O(b^{d+1})$
- Space complexity (bad)
  - Back pointers for all generated nodes $O(b^{d+1})$
  - The queue / fringe (smaller, but still $O(b^{d+1})$)

Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$

b: branching factor d: goal depth
C*: cost of optimal solution m: maximum depth of tree

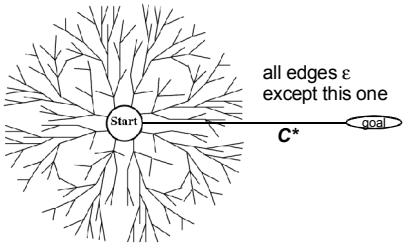
1: complete if b is finite
2: complete if step costs ≥ ε > 0
3: optimal if step costs are all identical
4: if both directions use breadth-first search

Uniform-cost search

- Find the least-cost goal
- Each node has a path cost from start. Expand the least cost node first.
- Use a priority queue instead of a normal queue
  - Always take out the least cost item
  - Remember *heap*? time ***O(log***(items in heap))

Uniform-cost search (UCS)

- Complete and optimal (if edge costs $\geq \varepsilon > 0$)
- Time and space: can be much worse than BFS
  - Let C^* be the cost of the least-cost goal
  - $O(b^{C^*/\varepsilon})$, possibly $C^*/\varepsilon >> d$



Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$
Uniform Cost Search	Yes[1,2]	Yes	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

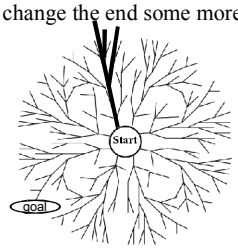
b: branching factor d: goal depth
C*: cost of optimal solution m: maximum depth of tree

1: complete if b is finite
2: complete if step costs $\geq \varepsilon > 0$
3: optimal if step costs are all identical
4: if both directions use breadth-first search

Depth-first search

1. Expand the deepest node first
2. Slightly change the end
3. Select a direction, go deep to the end
4. Slightly change the end some more…

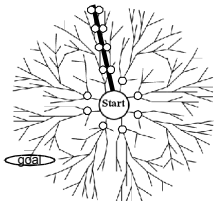
fan



fringe is a stack

What's in the fringe for DFS?

- m = maximum depth of graph from start
- ***m(b-1) ~ O(mb)***
- (Space complexity)

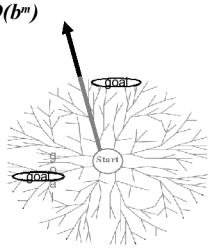


c.f. BFS $O(b^d)$

- "backtracking search" even less space
  - generate siblings (if applicable)

What's wrong with DFS?

- Infinite tree: may not find goal (incomplete)
- May not be optimal
- Finite tree: may visit almost all nodes, time complexity ***O(b^m)***



c.f. BFS ***O(b^d)***

Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$
Uniform Cost Search	Yes[1,2]	Yes	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
Depth First Search	No	No	O(bm)	$O(b^m)$

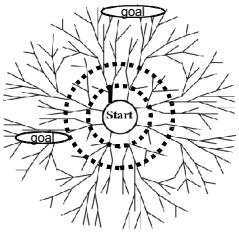
b: branching factor d: goal depth
C*: cost of optimal solution m: maximum depth of tree

1: complete if b is finite
2: complete if step costs ≥ ε > 0
3: optimal if step costs are all identical
4: if both directions use breadth-first search

How about this?

1. DFS, but stop if depth > 1.
2. If goal not found, repeat DFS, stop if depth >2.
3. And so on…

fan within ripple



Iterative deepening

- Search proceeds like BFS, but fringe is like DFS
  - Complete, optimal like BFS
  - Small space complexity like DFS
- A huge waste?
  - Each deepening repeats DFS from the beginning
  - No! ***$db+(d-1)b^2+(d-2)b^3+\ldots+b^d \sim O(b^d)$***
  - Time complexity like BFS
- Preferred uninformed search method

Defining the Environment

- State space : all valid configurations
- Initial (Start) and Goal states (can have more than one)
- Successor function : *successors*(**s**) are all states that can be reached from **s** by some legal action
- Cost : the cost to take an action from a state

This information can be represented as a weighted, directed graph

General Search Algorithm

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

General Search Algorithm with Uniform Cost Search

fringe←insert(*initial_nodes*)
loop do
  If empty(*fringe*) **then return** failure
  node←remove_first(*fringe*)
  if goalTest(*node*) **then return** solution(*node*)
  fringe←insert_all(expand(*node*))

Remember to Update Cost of Nodes on Fringe!

Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$
Uniform Cost Search	Yes[1,2]	Yes	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
Depth First Search	No	No	O(bm)	$O(b^m)$
Iterative Deepening	Yes[1]	Yes[3]	O(bd)	$O(b^d)$

b: branching factor
C*: cost of optimal solution
d: goal depth
m: maximum depth of tree

1: complete if b is finite
2: complete if step costs ≥ ε > 0
3: optimal if step costs are all identical
4: if both directions use breadth-first search

Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$
Uniform Cost Search	Yes[1,2]	Yes	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
Depth First Search	No	No	O(bm)	$O(b^m)$
Iterative Deepening	Yes[1]	Yes[3]	O(bd)	$O(b^d)$

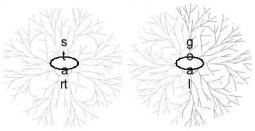
b: branching factor d: goal depth
C*: cost of optimal solution m: maximum depth of tree

1: complete if b is finite
2: complete if step costs ≥ ε > 0
3: optimal if step costs are all identical
4: if both directions use breadth-first search

How to Reduce Time Complexity

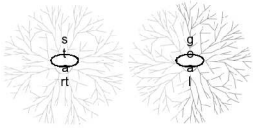
Bidirectional search

- Breadth-first search from both start and goal
- Fringes meet
- Generates $O(b^{d/2})$ instead of $O(b^d)$ nodes



Bidirectional search

- But
  - The fringes are $O(b^{d/2})$
  - How do you start from the 8-queens goals?



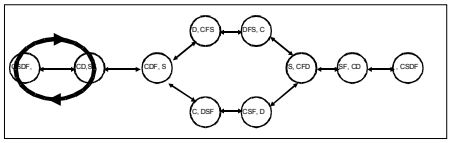
Performance of search algorithms on trees

	Complete	Optimal	Space	Time
Breadth First Search	Yes[1]	Yes[3]	$O(b^{d+1})$	$O(b^{d+1})$
Uniform Cost Search	Yes[1,2]	Yes	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
Depth First Search	No	No	O(bm)	$O(b^m)$
Iterative Deepening	Yes[1]	Yes[3]	O(bd)	$O(b^d)$
Bidirectional Search	Yes[1,4]	Yes[3,4]	$O(b^{d/2})$	$O(b^{d/2})$

b: branching factor d: goal depth
C*: cost of optimal solution m: maximum depth of tree

1: complete if b is finite
2: complete if step costs ≥ ε > 0
3: optimal if step costs are all identical
4: if both directions use breadth-first search

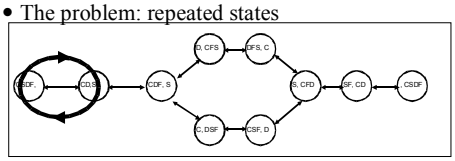
If state space graph is not a tree



Algorithms that forget their history are doomed to repeat it

If state space graph is not a tree

- The problem: repeated states



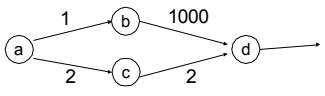
- Pretend it's a tree: wasteful (BFS) or impossible (DFS). Can you see why?
- How to prevent it?

If state space graph is not a tree

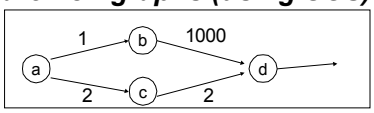
- We have to remember already-expanded states (CLOSED).
- BFS:
  - If we generate a state that's in CLOSED, don't add it to the queue – throw it away.
  - Still $O(b^{d+1})$ space complexity, not worse
- DFS:
  - If we generate a state that's in CLOSED, don't add it to the stack – throw it away.
  - Space was $O(bm)$, now $O(b^{d+1})$ – much worse!

Uniform Cost Search

- If you detect that you've generated an existing state, what do you do?
  - throw it away (BFS, DFS)
  - Is it the right thing for uniform-cost search?



Search on graphs (using UCS)



- The fringe (priority queue)

c finds out that d can be reached with a cheaper path

{(a,0)}
{(b,1), (c,2)}
{(c,2), (d,1001)}
{(d,4)}

- Needs to update the cost of existing states if better path found.

General Search (with closed list)

```
OPEN   = { startNode } // Nodes under consideration.
CLOSED = { }          // Nodes we're done with.

while OPEN is not empty
{
 remove an item from OPEN based on search strategy used
 - call it X

 if goalState?(X) return the solution found

 otherwise // Expand node X.
 {
    1) add X to CLOSED
    2) generate the immediate neighbors (ie, children of X)
    3) eliminate those children already in OPEN or CLOSED
    4) add REMAINING children to OPEN
 }
}
return FAILURE  // Failed if OPEN exhausted without a goal being found.
```

General Search (with closed list)

Note: Things may need to be **tailored a bit to each different search method** being used.

For example, in the iterative deepening search a node may be placed on the closed list because it is at the maximum depth for an iteration, but later the code may find another way to that node that would make the node less deep, and so the node would need to be moved out of the closed list and put back in the open list.

Another example, in uniform cost search you may find a less costly way of reaching a node that is already in the open list. You would then need to update the cost of the node in the open list.

Recap

- Uninformed search
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening
  - Bidirectional search
- Performance measures
  - Complete
  - Optimal
  - Time complexity
  - Space complexity
- Searching with Repeated States—keeping a closed list