o Time complexity
o Space complexity
• Correctness
o An algorithm is complete if it will find a solution when one exists and correctly report failure (no solution) when it does not
o An algorithm is optimal if it finds a solution with the lowest path cost among all solutions (i.e., path cost optimal)

Actions: si → A
Function that returns the possible actions, A {a1,..., ak}, at a given state si
Initial state

Transition model, T: (si, aj) → si'
Function that returns the intermediate state transitioned to, si', when action aj is applied at state si

Goal test, isGoal: si → {0, 1}
Function that returns 1 if given state si is a goal state, else returns 0

Action costs, cost: (si, aj, si') → v
Function that returns cost, v, of taking the action aj at state si to reach the intermediate state si'
Generally assume costs >= 0

general search algo

```
Function TreeSearch(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}        # AUT: Node(current_state, parent_node)
   while frontier not empty:                     # add path cost, depth, action as required
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               frontier.push(successor)
   return failure        # i.e., no path from the initial state to any goal state
```

BFS

Frontier: Queue

Time Complexity: O(b^d)

Space Complexity: O(b^d)

Complete: Yes[1]

Optimal: No[2]

b: branching factor
d: depth of shallowest goal
m: maximum depth
1: if finite b AND (finite m OR contains solution)
2: optimal if action costs uniform (and some other cases)

Default assumptions on search spaces:
- b finite
- d finite (contains a solution)
- m infinite
- All actions costs ≥ 0

Exercise questions:
- Show that when b is infinite, BFS is incomplete.
- Show that when b is finite, but m is infinite AND there is no solution, BFS is incomplete.
- Under what (other) cases is BFS optimal?
- Why is the complexity of BFS O(b^d)? Is this accurate?

Early goal test for BFS:

Performing goal test on pushing to frontier instead of popping from frontier will prevent this with no change in the expectation on the BFS solution

However, it should be noted that a different (valid) path may be output

UCS:

Frontier: Priority Queue[1]

Time Complexity: O(b^{1+⌊C*/ε⌋})

Space Complexity: O(b^{1+⌊C*/ε⌋})

Complete: Yes[2]

Optimal: Yes

Updated default assumptions on search spaces:
- All action costs > ε > 0

Exercise questions:
- Can you show that UCS under BFS completeness assumptions is incomplete when action costs are not > ε > 0 (e.g., when ε = 0)?
- Why is the complexity of UCS O(b^{1+⌊C*/ε⌋})?
- Why is UCS optimal?
- Would UCS still be optimal with an early-goal test?

b: branching factor
ε: 1 + ⌊C*/ε⌋, where C* is the optimal path cost and ε is some small positive constant
1: prioritising lower path cost, g(n), where g(n) = path cost of the path taken to reach n
2: requires BFS completeness criteria and that action costs > ε > 0
Note: determining path cost for each node is O(1) since we store the current path cost in each node

DFS:

Frontier: Stack

Time Complexity: O(b^m)

Space Complexity: O(bm)

Complete: No[1]

Optimal: No

Exercise questions:
- Show that DFS under BFS completeness assumptions is incomplete.
- Can space complexity be improved?

The space complexity of DFS may be improved to O(m) by simply backtracking – i.e., tracing back to parent and last action taken (assuming fixed order of actions – recall that we store parent node and action taken at parent)

b: branching factor
m: maximum depth
1: DFS may be incomplete even if a solution exists

Depth-Limited Search (DLS)

DFS with a depth limit, ℓ
- Search only up to depth ℓ
- Assume no actions may be taken from nodes at depth ℓ

Same guarantees as DFS with ℓ in place of m
- Time complexity: O(b^ℓ)
- Space complexity: O(bℓ)
- Complete: No
- Optimal: No

Iterative Deepening Search (IDS)

Example, b = 10 and d = 5
- N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
- N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
- Overhead ≈ 11%

IDS properties
- Time: O(b^d)
- Space: O(bd)
- Complete: Yes (is b finite and d finite or contains solution) – same as BFS
- Optimal: No (optimal if costs uniform (and some other cases)) –same as BFS

Summary tree search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	No[4]	Yes	No	No	No[4]
Time	O(b^d)	O(b^{1+⌊C*/ε⌋})	O(b^m)	O(b^ℓ)	O(b^d)
Space	O(b^d)	O(b^{1+⌊C*/ε⌋})	O(bm)	O(bℓ)	O(bd)

Default assumptions on search spaces:
- b finite
- d finite (contains solution)
- m infinite
- All actions costs are > ε > 0

1. Complete if b finite and either has a solution or m finite
2. Complete if all actions costs are > ε > 0
3. DFS is incomplete unless the search space is finite – i.e., when b finite and m finite
4. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test is assumed for BFS
- UCS must perform a Late Goal Test to be optimal (this also accounts for the +1 in the index of its complexity)
- DFS is incomplete (even under 1 – note the "or"); if a solution exists, it may infinitely traverse a path without a solution
- DFS space complexity may be improved to O(m) with backtracking (similar for DLS and IDS)

Tree search allows revisits, cyclic graph, incomplete

graph search V1

```
Function GraphSearchV1(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached:
                     frontier.push(successor)
                     reached.insert(successor.state: successor)
   return failure
```

Exercise:
- Which uninformed algorithms may benefit from this implementation?

Not optimal bc cannot revisit even with more optimal path

graph search V2

```
Function GraphSearchV2(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached or
                  successor.getCost() < reached[successor.state].getCost():
                     frontier.push(successor)
                     reached.insert(successor.state: successor)
   return failure
```

Summary graph search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	No[4]	Yes	No	No	No[4]
Time	O(|V|+|E|)				
Space					

Default assumptions on search spaces:
- b finite
- d finite (contains solution)
- m infinite
- All actions costs are > ε > 0

1. Complete if b finite and either has a solution or m finite
2. Complete if all actions costs are > ε > 0
3. DFS is incomplete unless the search space is finite – i.e., when b finite and m finite
4. Cost optimal if action costs are all identical (and several other cases)

- Time and space complexities are now bounded by the size of the search space – i.e., the number of vertices (nodes) and edges, |V| + |E|
- Note that we do not need to check for cheaper paths under graph search for BFS and DFS since costs play no part in those algorithms and they cannot guarantee an optimal solution anyway

For CS3243, unless otherwise mentioned, assume tree search

For graph search, assume V2 for UCS and V1 for other uninformed search algorithms

UCS optimal

Assumptions
- Branching factor (b) is finite
- Solution exists (d finite), or max depth (m) is finite
- Each action cost exceeds a small positive constant, ε

} Assert that UCS is complete!

Proposition
- Suppose UCS finds a suboptimal path T with goal state n_T
- That implies a different optimal path U with goal state n_U (note n_U may be the same state as n_T)
  - This means that node n_T (via T) was popped before some other node n' along $U \Rightarrow g(n_T) < g(n')$
  - Since both U and T start from the same initial state, sub-paths of U and T must be on the frontier
  - The above states that the last node along U that was not popped is n', and n_T was popped instead
- Thus, we have:
  - $g(n') < g(n_T)$ // assert minimum action cost $\epsilon > 0$
  - $g(n_U) < g(n_T)$ // definition of optimal path
  - $g(n') < g(n_T)$

BFS tracing example

Node	Frontier (push/pop from right to left)
S()	A(S), B(S)
B(S)	A(S), C(S,B), F(S,B)
F(S,B)	A(S), C(S,B), D(S,B,F), H(S,B,F)
H(S,B,F)	A(S), C(S,B), D(S,B,F), D(S,B,F,H), K(S,B,F,H)
K(S,B,F,H)	A(S), C(S,B), D(S,B,F), D(S,B,F,H)
D(S,B,F,H)	A(S), C(S,B), D(S,B,F), G(S,B,F,H,D)
G(S,B,F,H,D)	PATH FOUND = S-B-F-H-D-G

- Common mistakes
  - Specifying nodes explored instead of the final path
  - Excluding node H from the path

Tree vs grapph search
- graph-search algorithms will only explore non-redundant paths
- only explores nodes that either have not been visited or have been visited but previously via less optimal paths

- tree search algorithms have no such restrictions; they consider all paths, including redundant ones

Admissible
paths not ending at a goal are never over estimated
- non goal, n, f(n) = g(n) + h(n) <= g(n) + h*(n)
paths ending at a goal are exact
- goal m, f(m) = g(m) + h(m), where h(m) = 0
if admissible, then A* optimal with tree search

Consistency
h(n) is consistent, if ∀n, n': h(n)≤ cost(n,a,n')+ h(n')
if h(n) is consistent, A* with graph search is optimal

If, h1(n)≥ h2(n), then h1 dominates h2
If h1 is also admissible
– h1 must be closer to h* than h2
– h1 is usually more efficient than h2 when used with A*

consistent -> admissible

Graph search V3

```
Function GraphSearchV3(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}
   reached = {}
   while frontier not empty:
         current = frontier.pop()
         reached.insert(current.state: current)
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached:
                     frontier.push(successor)
   return failure
```

b. Show graph search for Greedy Best-First Search is complete (assuming finite search space).
Proof Sketch:
– When a node is expanded, each successor:
  —is added to frontier, or
  —has already been visited (graph search).
– Implies that at least one path to each state in the search space will eventually be added to the frontier (finite search space).
– So, a goal, if it exists must be found, or else the algorithm terminates when the frontier is empty (all states explored).

Let s be the initial state, n be an intermediate state along the optimal path, t be a suboptimal goal state (i.e., a goal state reached via a suboptimal path), and t* be the goal along the optimal path.
An optimal solution implies that n must be expanded before t.
Proof by contradiction:
• Let us assume that a suboptimal solution is found – i.e., that t is expanded before n, which implies that (A): f(t) ≤ f(n).
• In other words, given the above frontier, only when f(t) < f(n) would we expand t before n.
• However, since t is not on the optimal path but t* is, we have:
o f(t) > f(t*)
o f(t) > g(t*) // since h(t*) = 0
o f(t) > g(n) + p(n, t*) // where p(n, t*) is the actual cost from n to t*
o f(t) > g(n) + h(n) // asserting admissibility
o f(t) > f(n) // this contradicts (A)
• Note: we do not consider f(t) = f(n) since that would mean f(t) is equally optimal.

Prove that A* Search is optimal under graph search v3 with consistent heuristic.
o Base case: When expanding the start node, it is obvious that $g\ s0 = g*\ s0$
o Inductive case: Assume the inductive hypothesis is true up to sk.
So, from $s0$ to sk, the optimal path is found, and each $g\ si$ Since $g*\ sk{+}1$ is the optimal path cost from $s0$ to $sk{+}1$,
we have: $\boldsymbol{g\ (sk{+}1) \geq g*\ (sk{+}1)}$– (A)
By consistency, we know that sk must be popped before $sk{+}1$.
So, after sk is popped, we know that when $sk{+}1$ is popped,
cost to node $sk{+}1$ is the minimum of all the ways that it has been reached so far (asserting priority queue based on f).
$\boldsymbol{g\ (sk{+}1)}$ + h(sk+1) = *MIN* (g(sk+1) + h(sk+1), g(sk) + c(sk, sk+1) + h(sk+1))
$\boldsymbol{g\ (sk{+}1)}$ = min{g(sk+1), g(sk) + c(sk, sk+1)}
≤ $g\ (sk) + cost\ (sk,\ sk{+}1)$
= $g*\ (sk) + cost\ (sk,\ sk{+}1)$
= $\boldsymbol{g*\ (sk{+}1)}$ – (B)

Solution: The proof is by induction of k(n), which denoted the number of actions required to reach the goal from a node n to the goal node t.
Base case (k = 1, i.e., the node n is one step from t): Since the heuristic function h is consistent, h(n) ≤ c(n, t) + h(t). And since h(t) = 0, h(n) ≤ c(n, t) = h*(n). Therefore, h is admissible.
Induction case: Suppose that our assumption holds for every node that is k – 1 actions away from t, and let us observe a node n that is k actions away from t; that is, the least-actions optimal path from n to t has k > 1 steps.
We write the optimal path from n to t as: n → n1 → n2 → … → nk–1 → t.
Since h is consistent, we have h(n) ≤ c(n, n1) + h(n1).
Now, note that since n1 is on a least-cost path to t from n, we must have that the path n1 → n2 → … → nk–1 → t is a minimal-cost path from n1 to t as well. By our induction hypothesis, we have h(n1) ≤ h*(n1).
Consequently, combining the two inequalities above, we have, h(n) ≤ c(n, n1) + h*(n1).
Note that h*(n1) is the cost of the optimal path from n1 to t; by our previous observation (that n1 → n2 → … → nk–1 → t is an optimal cost path from n1 to t), we have that the cost of the optimal path from n to t – i.e., h*(n) – is exactly c(n, n1) + h*(n1), which concludes the proof.

admissibility but inconsistent
state an admissible heuristic that overestimates the path cost from parent to child (happens when child is understated too much, so h(n) > cost(n to n') + h(n') (h(n') understated)

Completeness assumptions:
- b finite AND (m finite OR has a solution)
- All action costs are > ε > 0

- UCS
  - Optimal under
    - Tree search
    - Graph search (version 2 and 3)
  - Popping node n ⇒ optimal path to n found
- Greedy Best-First Search
  - Not optimal
  - Incomplete under tree search

Assume graph search version 1 for Greedy (unless otherwise stated)

- A* Search
  - Assuming admissible h
    - May not have monotonically increasing path cost
    - Optimal under
      - Tree search
      - Graph search (version 2)
  - Assuming consistent h
    - On popping node n, optimal path to n found
    - Optimal under
      - Tree search
      - Graph search (version 2 and 3)

Assume graph search version 2 and late goal testing for UCS and A* (unless otherwise stated)

h_1	h_2	$MAX(h_1, h_2)$
Admissible	Admissible	Admissible
Admissible	Inadmissible	Inadmissible
Inadmissible	Inadmissible	Inadmissible

h_1	h_2	$MIN(h_1, h_2)$
Admissible	Admissible	Admissible
Admissible	Inadmissible	Admissible
Inadmissible	Inadmissible	Uncertain

What about consistent heuristics?
- MAX of 2 consistent heuristics is also consistent.
- MIN of 2 consistent heuristics is also consistent.

Try to determine these as an exercise! Might appear in the midterm!

h_1: Number of pallets left. Admissible.
- Every move made by the Pacman eats 0 or 1 pallets ⇒ Moves required to clear all pallets would be greater than or equal to the number of pallets on the board
  - i.e., Pacman requires n moves to clear n pallets ∴ Pacman eats 1 pallet per move.

h_2: Number of pallets left + Manhattan distance to the closest pallet. Inadmissible, by counterexample.
- Consider the case where there is 1 pallet remaining, and Pacman is 1 step away from that pallet.
- Here, we have: $h_2 = 2$ (distance 1 and 1 pallet left), whereas $h^* = 1$. Thus, h_2 overestimates.

h_3: Manhattan distance to the furthest pallet. Admissible.
- To consume that furthest pallet, Pacman must first travel to that pallet, which requires at least Manhattan distance number of moves.
- Note that a common mistake is to begin the proof claiming that $h_3 = h^*$ when only 1 pallet remains on the board.
  - This is untrue since there may still be obstacles on the board – i.e., we instead have $h_3 \leq h^*$.

h_4: Average Euclidean distance to all pallets. Admissible.
- Euclidean distance ≤ Manhattan distance AND average distance ≤ maximum distance ⇒ h_3 dominates h_4
- Thus, since h_3 is admissible, then h_4 is also admissible.

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

Stochastic hill climbing
- Changes the function highest_valued_successor(..)
- Chooses randomly among states with f-values better than current
- May take longer to find a solution but sometimes leads to better solutions

First-choice hill climbing
- Changes the function highest_valued_successor(..)
- Handles high b by randomly generating successors until one with better f-value than current is found (instead of generating all possible successors)
- May even work with infinite b

Sideways move
- Replaces ≤ with <
- Allows continuation when eval_fn(neighbour) == eval_fn(current)
- Can traverse shoulders / plateaus

Random-restart hill climbing
- Different algorithm
- Adds an outer loop which randomly picks a new starting state
- Keeps attempting random restarts until a solution is found

Local Beam Search
Store k states instead of 1
o Hill climbing just stores the current state
o Beam (window) stores k
• General algorithm
o Begins with k random starts
o Each iteration generates successors for each of the k random start states
o Repeat with best k among ALL generated successors unless goal found
• Better than k parallel random restarts
o Since best k among ALL successors taken (not best from each set of successors, k times)
• Stochastic beam search
o Original variant may still get stuck in a local cluster
o Adopt stochastic strategy similar to stochastic hill climbing to increase state diversity

```
Function BeamSearch(random_initial_state, isGoal, actions, T, choose_best_k):
   frontier = {}
   for i in 1 to k:
      s = random_initial_state()
      if isGoal(s): return s
      frontier.push(s)
   while True:
      new_frontier = {}
      while frontier not empty:
         s = frontier.pop()
         for a in actions(s):
            new_s = T(s, a)
            if isGoal(new_s): return new_s
            new_frontier.push(new_s)
      frontier = choose_best_k(new_frontier)
```

As the search may loop infinitely, it may be wise to define and use an iteration threshold

No duplicate states on beam

We may also apply stochastic beam search where instead of choosing the best k, it chooses successors with probability proportional to the successor's value

- Notice that we may allow states with lower values than their parents to be added since choose_best_k will simply choose the best k successor states generated
  - We may vary this rule but then may not be able to always ensure we have k new candidates