Fall 2024

BECE309L – AI & ML

Problem Solving by Searching

Module – 2

Dr. Nitish Katal

Outline

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search

Agents that Plan

Reflex Agents

- Reflex agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Consider how the world IS

Demo of Reflex Agent

Optimal Scenario:

Real Scenario:

Planning Agents

- Planning agents:
  - Ask “what if”
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Must formulate a goal (test)
  - Consider how the world WOULD BE
- Optimal vs. complete planning
- Planning vs. replanning

Demo: Planning Agent

Replanning

Optimal Planning

Basic Terminologies

- **Search:**
  - Searching is a step by step procedure to solve a search-problem in a given search space.
  - In a ***problem-solving context***, the state space represents all possible states that a system or agent can be in.
  - Each state represents a particular configuration, situation, or condition.

Basic Terminologies

- **States:**
  - In the context of a state space, a "state" refers to a specific configuration or situation that the problem-solving agent can occupy.
  - States can represent a wide range of conditions, depending on the problem.

Basic Terminologies

- A search problem can have three main factors:
  - **Search Space**: Search space represents a set of possible solutions, which a system may have.
  - **Initial State**: It is a state from where agent begins the search.
  - **Goal test/state**: The desired state that the system or agent aims to reach. The goal state is typically defined in terms of certain criteria or conditions.
- **Search tree**: A tree representation of search problem. The root of the search tree is the root node which is corresponding to the initial state.

Basic Terminologies

- **Actions:**
  - It gives the description of all the available actions to the agent.
- **Path:**
  - A path is a sequence of actions or operators that leads from the initial state to a particular state in the search space.

Basic Terminologies

Transition model:

A description of what each action do, can be represented as a transition model.

Search Algorithm:

Forward state space search is guided by a search algorithm that systematically explores the search space, searching for a solution path from the initial state to a goal state.

Path Cost:

It is a function which assigns a numeric cost to each path.

Solution:

It is an action sequence which leads from the start node to the goal node.

Optimal Solution:

If a solution has the lowest cost among all solutions.

Heuristic Functions:

May employ heuristic functions to guide the search more efficiently by estimating the cost or distance to the goal state from a given state.

Informed and Uninformed Search:

Informed (e.g., A*) use heuristic information,

While uninformed search algorithms (e.g., BFS, DFS) operate without heuristic guidance.

Complex Problem-Solving:

State space search aids in solving complex problems, optimizing processes, and improving decision-making in various real-world scenarios.

Search Problems

- A search problem consists of:
  - A state space

  - A successor function (with actions, costs)

  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

What's in a State Space?

The world state includes every last detail of the environment

A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END
- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?

    $120 \times (2^{30}) \times (12^{2}) \times 4$
  - States for pathing?

    120
  - States for eat-all-dots?

    $120 \times (2^{30})$

State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea

Tiny state space graph for a tiny search problem

Search Trees

- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees

Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.

State Space Graphs vs. Search Trees

Consider this 4-state graph:

How big is its search tree (from S)?

Important: Lots of repeated structure in the search tree!

Properties of Search Algorithms

Completeness:	A search algorithm is said to be complete **if it guarantees to return a solution** if at **least any solution exists** for **any random input**.
Optimality:	If **a solution found for an algorithm** is **guaranteed** to **be the best solution** (lowest path cost) **among all other solutions.**
Time Complexity:	Is a measure of **time** for an **algorithm** to **complete** its **task**.
Space Complexity:	It is the **maximum storage space required** at any **point during** the **search**, as the complexity of the problem.

Properties of Search Algorithms

Completeness:	A search algorithm is said to be complete **if it guarantees to return a solution** if at **least any solution exists** for **any random input**.
Optimality:	If **a solution found for an algorithm** is **guaranteed** to **be the best solution** (lowest path cost) **among all other solutions.**
Time Complexity:	Is a measure of **time** for an **algorithm** to **complete** its **task**.
Space Complexity:	It is the **maximum storage space required** at any **point during** the **search**, as the complexity of the problem.

Types of Searches

Types of Searches

Search Example: Romania

Searching with a Search Tree

- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

- Important ideas:
  - Fringe
  - Expansion
  - Exploration strategy
- Main question: which fringe nodes to explore?

Example: Tree Search

Example: Tree Search

~~s~~

~~s → d~~

s → e

s → p

s → d → b

s → d → c

~~s → d → e~~

s → d → e → h

~~s → d → e → r~~

~~s → d → e → r → f~~

s → d → e → r → f → c

~~s → d → e → r → f → G~~

1. Uninformed/Blind Search

- Does not contain any domain knowledge such as closeness, the location of the goal.
- Operates in a brute-force way.
  - As it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal.
- It examines each node of the tree until it achieves the goal node.

1. Uninformed/Blind Search

- **Advantages :**
  - **Simplicity**: Uninformed search algorithms can be likened to straightforward puzzle-solving. Their simplicity makes them an excellent starting point for beginners, as they *don't involve complex strategies or pre-existing knowledge*.
  - **Efficiency in Simplicity**: These algorithms *excel* when dealing *with less complex problems*. They **swiftly reach solutions**, particularly when the search space isn't vast or complicated.
  - **Versatility**: Comparable to a toolkit containing various tools, uninformed search algorithms possess the adaptability to tackle different challenges. Whether deciphering puzzles or navigating perplexing mazes, these algorithms rise to the occasion.

- **Challenges :**
  - Inefficiency at Times: Visualize these algorithms as *explorers occasionally taking scenic detours*. In situations of complexity, these detours can slow down the search process.
  - Getting Stuck: *Picture a maze with concealed dead ends*. Similarly, uninformed search algorithms can become trapped where they can't advance. It leads to moments of frustration and inefficiency.

Depth-First Search

Depth First Search

Strategy: *expand a deepest node first*

Implementation: *Fringe is a LIFO stack*

Search Algorithm Properties

Depth-First Search (DFS) Properties

- DFS is a **recursive algorithm** for traversing a tree or graph data structure,
  - Which **starts** from the **root node** and
  - Follows **each path to its greatest depth node** before moving to the next path.
- DFS uses a **stack data structure** for its implementation.
- Completeness: DFS search algorithm is **complete within finite state space** as it will expand every node within a limited search tree.
- Time Complexity:
  - Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$
  - Where, m= maximum depth of any node and this can be much larger than b (Shallowest solution depth)
- Space Complexity:
  - DFS algorithm needs to store **only single path from the root node**, hence space complexity of DFS is equivalent to the size of the fringe set, which is
  - O(bm).

Search Algorithm Properties

- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Figure of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots. b^m = O(b^m)$

Depth-First Search (DFS) Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent that
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost

Depth-First Search Properties

- Advantages:
  - DFS **requires very less memory** as it only needs to store a stack of the nodes on the path from root node to the current node.
  - It **takes less time to reach to the goal node** than BFS algorithm (if it traverses in the right path).
- Disadvantages:
  - There is the **possibility that many states keep re-occurring**, and **there is no guarantee of finding the solution**.
  - DFS algorithm goes for **deep down searching** and sometime it **may go to the infinite loop**.

Breadth-First Search

Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

Breadth-First Search

- A strategy for *traversing a tree or graph*, which searches *breadthwise* in a *tree* or *graph*.
- BFS algorithm **starts searching** from the **root node** of the tree and **expands** all **successor node** at the **current level before moving** to **nodes** of next level.
- An example of a **general-graph search algorithm**.
- BFS is implemented using **First-In-First Out (FIFO) queue data structure**.

- **Time Complexity:**
  - Can be obtained by the number of nodes traversed in BFS until the shallowest Node.
  - $1 + b + b^2 + \ldots. b^s = O(b^s)$
  - Where b: Branching factor (maximum number of successors of any node).
  - s: Depth of the shallowest goal node.
- **Space Complexity:**
  - Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - Only if costs are all 1

Breadth-First Search Properties

- Advantages:
  - BFS will **provide** a **solution** if **any solution exists**.
  - If there are **more than one solutions** for a given problem, then BFS will provide the **minimal solution** which requires the **least number of steps**.

- Disadvantages:
  - It **requires lots of memory** since each level of the tree must be saved into memory to expand the next level.
  - BFS **needs lots of time** if the solution is far away from the root node.

DFS v/s BFS

Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).

Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Uniform Cost Search

Cost-Sensitive Search

BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path.

Uniform Cost Search

Strategy: *expand a cheapest node first*:

Fringe is a priority queue (priority: cumulative cost)

Uniform Cost Search (UCS) Properties

- A searching algorithm used for **traversing a weighted tree** or graph.
  - Used when a **different cost** is **available** for **each edge**.
- **Primary Goal:** to find a path to the goal node which has the lowest cumulative cost.
- UCS **expands** nodes according to **their path costs** form the **root node**.
  - It can be used to solve any graph/tree where the optimal cost is in demand.
- UCS algorithm is implemented by the **priority queue**. It gives **maximum priority** to the **lowest cumulative cost**.

Algorithm

- Set a variable NODE to the initial state, i.e., **the root node** and expand it.
- **After expanding** the **root node,**
  - **Select one node** having the **lowest path cost** and
  - Expand it further.
  - *(Remember, the selection of the node should give an optimal path cost.)*
- If the goal node is searched with optimal value, return goal state, else carry on the search.

Uniform Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)

- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)

Uniform Cost Search (UCS) Properties

- **Completeness**: It guarantees to reach the goal state.
- **Optimality**: It gives optimal path cost solution for the search.
- **Space and time complexity**:
  - The worst space and time complexity of the uniform-cost search is
  - Let **C^*** is Cost of the optimal solution, and
  - **ε** is each step to get closer to the goal node.
  - Then the number of steps is = **$C^*/\varepsilon+1$**.
  - Here we have taken +1, as we start from state 0 and end to **C^*/ε**.
  - Hence, the worst-case time complexity of Uniform-cost search is **$O(b^{1 + [C^*/\varepsilon]})$**
  - The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is
  - **$O(b^{1 + [C^*/\varepsilon]})$**.
- **Optimal**: Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Uniform Cost Issues

- Remember: UCS explores increasing cost contours

- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location

Depth Limited Search

Depth Limited Search

- DLS algorithm is similar to DFS with a **predetermined limit**.
- DLS can solve the **drawback** of the **infinite path** in the DFS .
- Uses **LIFO** (Last in First Out) order, which is based on the stack, in **order** to **expand** the **unexpanded nodes** in the **search tree**.
- **Search proceeds** to the **deepest level** of the **tree where** it has **no successors**. This search expands nodes till infinity, i.e., the depth of the tree.
- In **DLS**, the **node** at the **depth limit** will **treat** as it has **no successor** nodes further.

DFS Algorithm

DFS works starting from the initial node **A** (root node) and traversing in one direction deeply till node **I** and then backtrack to B and so on. Therefore, the sequence will be **A->B->D->I->E->C->F->G.**

Algorithm

- Set a variable NODE to the initial state, i.e., the **root node.**
- Set a **variable GOAL** which contains the **value** of the **goal state.**
- **Loop each node** by **traversing deeply** in **one direction**/path in search of the **goal node.**
- While performing the looping, start **removing the elements** from the stack in **LIFO order.**
- If the **goal state** is **found, return goal state** otherwise **backtrack** to expand nodes in other direction.

DLS : Properties

- ***Completeness***: Is complete if the solution is above the depth-limit.
- ***Time Complexity***: Time complexity of DLS algorithm is $O(b^{\ell})$.
- ***Space Complexity***: Space complexity of DLS algorithm is $O(b \times \ell)$.
- ***Optimal***: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.
- ***Advantages***:
  - Depth-limited search is Memory efficient.
- ***Disadvantages***:
  - Incompleteness.
  - It may not be optimal if the problem has more than one solution.

Iterative Deepening First Depth Search

Iterative Deepening

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..

Iterative Deepening First Depth Search

- Performs **depth-first search** up to a **certain "depth limit",** and it **keeps increasing** the **depth limit** after **each iteration** until the **goal node is found**.
- **Combines** the **benefits** of **BFS's fast search** and **DFS's memory efficiency**.
- **Useful** in **uninformed search** when **search space** is **large**, and **depth** of **goal node** is **unknown**.
- IDS blends aspects of both BFS and DFS.
- It ***initiates with BFS*** and ***progressively probes deeper levels*** until a ***solution emerges***.
- While it **guarantees a solution**, **IDS might not be as efficient as pure BFS or DFS**.

1'st Iteration-----> A

2'nd Iteration----> A, B, C

3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

Iterative Deepening First Depth Search

- The performance measure of Iterative deepening search
- **Completeness**: Iterative deepening search **may or may not** reach the goal state.
- **Optimality**: It **does not** give an **optimal solution** always.
- **Space Complexity**: It has the **same space complexity** as BFS, i.e., **O(bd).**
- **Time Complexity**: It has **O(d)** time complexity.
- **Disadvantages**
  - it seems ***wasteful*** because it ***generates states multiple times***.

Bi-directional Search

Bi-directional Search

- Bidirectional search algorithm **runs two simultaneous searches**, ***one form initial state*** called as ***forward-search*** and ***other*** from ***goal*** node called as ***backward-search***, to ***find*** the ***goal node***.
- Bidirectional search **replaces one single search graph** with **two small subgraphs** in which **one starts** the search from **an initial vertex** and other starts from **goal vertex**.
- The **search stops** when these **two graphs intersect each other**.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Bi-directional Search

- **Completeness**: Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity**: Time complexity of bidirectional search using BFS is $O(b^d)$.
- **Space Complexity**: Space complexity of bidirectional search is $O(b^d)$.
- **Optimal**: Bidirectional search is Optimal.
- **Advantages**:
  - Bidirectional search is fast.
  - Bidirectional search requires less memory
- **Disadvantages**:
  - Implementation of the bidirectional search tree is difficult.
  - In bidirectional search, one should know the goal state in advance.

Summary

Criteria	BFS	Uniform-cost	DFS	Depth-limited	IDS	Bidirectional
Complete?	Yes[#]	Yes[#&]	No	No	Yes[#]	Yes[#+]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/e \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[$]	Yes	No	No	Yes[$]	Yes[$+]

b: Branching factor
d: Depth of the shallowest goal
l: Depth limit
m: Maximum depth of search tree
e: The lower bound of the step cost

[#]: Complete if *b* is finite
[&]: Complete if step cost >= *e*
[$]: Optimal if all step costs are identical
[+]: If both direction use BFS

Informed Search

Outline

- Informed Search
  - Heuristics
  - Greedy Search
  - A* Search

Recap: Search

- Search problem:
  - States (configurations of the world)
  - Actions and costs
  - Successor function (world dynamics)
  - Start state and goal test
- Search tree:
  - Nodes: represent plans for reaching states
  - Plans have costs (sum of action costs)
- Search algorithm:
  - Systematically builds a search tree
  - Chooses an ordering of the fringe (unexplored nodes)
  - Optimal: finds least-cost plans

Example: Pancake Problem

Cost: Number of pancakes flipped

Example: Pancake Problem

State space graph with costs as weights

General Tree Search

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Informed Search

Informed Search

- Such algorithms contain an ***array of knowledge*** such as ***how far we are from the goal***, ***path cost***, ***how to reach to goal node***, etc.
- **Heuristic Function:** finds the ***most promising path***.
  - It **estimates** how **close** a **state** is **to the goal**.
  - $\boldsymbol{h(n)}$ is the **heuristic cost**, and **calculates** the **cost** of an **optimal path between** the **pair** of **states**.
  - $\boldsymbol{h^*(n)}$ is the estimated cost.

$$\boldsymbol{h(n) \leq h^*(n)}$$

  - ***HF*** takes the **current state** of the **agent** as its **input** and **produces** the **estimation** of how close agent is from the goal.
- The heuristic method, however, *might not always give the best solution*, but it *guaranteed to find a good solution* in *reasonable time*.

Search Heuristics

- A heuristic is:
  - A function that ***estimates*** how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing

Example: Heuristic Function

Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Example: Heuristic Function

Heuristic: the number of the largest pancake that is still out of place

Greedy Search

Greedy Search

- Expand the node that seems closest…

- What can go wrong?

Greedy Search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state

- A common case:
  - Best-first takes you straight to the (wrong) goal

- Worst-case: like a badly-guided DFS

Greedy Search

- **Performance Measures:**
- ***Completeness***: Best-first search is ***incomplete*** even in finite state space.
- ***Optimality***: It **does not provide** an optimal solution.
- ***Time and Space complexity***: It has **$O(b^m)$** worst time and space complexity, where m is the maximum depth of the search tree. If the quality of the heuristic function is good, the complexities could be reduced substantially.
- ***Disadvantages of Best-first search***
  - BFS does **not guarantees** to reach **the goal state**.
  - Since the best-first search is a greedy approach, it **does not give an optimized solution**.
  - It may cover **a long distance** in some cases.

A* Search

A* Search

UCS

Greedy

A*

Uniform-Cost Search

Greedy Search

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)

- A* Search orders by the sum: f(n) = g(n) + h(n)

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)

- A* Search orders by the sum: f(n) = g(n) + h(n)

When should A* terminate?

- Should we stop when we enqueue a goal?

- No: only stop when we dequeue a goal

Properties of A*

Properties of A*

Uniform-Cost

A*

UCS vs A* Contours

- Uniform-cost expands equally in all "directions"

- A* expands mainly toward the goal, but does hedge its bets to ensure optimality

Comparison

Greedy

Uniform Cost

A*

A* Applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …

A* Performance

- **Completeness**: The star(*) in A* search **guarantees** to reach the goal node.
- **Optimality**: An underestimated cost will **always give an optimal solution**.
- **Space and time complexity**: A* search has O(bd) space and time complexities.
- **Advantage of A* search**
  - A* is **efficient** if an **efficient** and **acceptable heuristic function** is used.
  - A* can be used to find **optimal solutions efficiently** as long as **a meaningful heuristic can be defined**.
- **Disadvantage of A* search**
  - A* mostly **runs out of space** for a long period.

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems

Search and Models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your **search** is **only as good as** your **models**…

Search Gone Wrong?

Search Gone Wrong?

Reference: https://wikidocs.net/189098

Search in Complex Environments

Search in Complex Environments

- In search algorithm, When a goal is found, the path to that goal constitutes a solution to the problem.

Reference: https://wikidocs.net/189098

Local Search

Local search algorithms

- In many optimization problems, ***path*** is irrelevant; the goal state ***is*** the solution
- Then **state space** = **set of "complete" configurations**;
  find ***configuration satisfying constraints***, e.g., n-queens problem; or, find ***optimal configuration***, e.g., travelling salesperson problem

- In such cases, can use ***iterative improvement*** algorithms: keep a single "current" state, try to improve it
- Constant space, suitable for online as well as offline search

Heuristic for *n*-queens problem

- **Goal**: n queens on board with no ***conflicts***, i.e., no queen attacking another
- **States**: n queens on board, one per column
- **Actions**: move a queen in its column
- **Heuristic value function**: number of conflicts

h = 5 h = 2 h = 0

Local Search Algorithms

- Operate by ***searching*** from a ***start state*** to ***neighboring states***, **without keeping track** of the **paths**, nor **the set of states** that have been reached.
- Local search algorithms can also **solve optimization problems**, in which the aim is to **find the best state** according to an **objective function**.
- They are **not systematic** – they **might never explore a portion** of the **search space** where a **solution actually** resides.
- **Two advantages:**
  - Use very little memory, and
  - Often find reasonable solutions in large or infinite state spaces (systematic algorithms may be unsuitable).

Different Regions in a State Space Diagram

Local maximum:

- It is a state which is **better than its neighboring state** however there exists a state which is better than it(global max.).
- This state is better because here the value of the **objective function is higher** than its neighbors.

Global maximum:

- It is the **best possible state** in the state space diagram. This is because, at this stage, the objective function has the highest value.

Plateau/flat local maximum:

- It is a **flat region of state space** where **neighboring states** have the **same value**.

Ridge:

- It is a **region** that is **higher than** its **neighbors** but **itself has a slope**. It is a special kind of local maximum.

Current state:

- The region of the state space diagram where **we are currently present** during the search.

Shoulder:

- It is a **plateau** that has an **uphill edge**.

Global and local maxima

Random restarts

- find global optimum

Random sideways moves

- Escape from shoulders
- Loop forever on flat local maxima

Hill Climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

Hill Climbing

- The **algorithm starts** with an **initial solution** and then **iteratively makes small changes** to it in **order** to **improve** the **solution**.
- Algo. **keeps track** of **one current state** and **on each iteration, moves** to the **neighboring state** with **highest value** — that is, it *heads in the direction that provides the steepest ascent*.
- **Changes** are **based** on a **heuristic function** that **evaluates** the **quality** of the **solution**.
- **Algo. continues** to **make** these **small changes** until it **reaches** a **local maximum**, meaning that no further improvement can be made with the current set of moves.
- It **terminates** when it reaches a "**peak**" where **no neighbor** has a **higher value**.
- Hill climbing **does not look ahead beyond** the **immediate neighbors** of the current state.

Hill Climbing

Key Features:

- **Generate and Test Approach:**
  - This feature involves **generating neighboring solutions** and **evaluating their effectiveness**, always aiming for an upward move in the solution space.
- **Greedy Local Search:**
  - The algorithm uses a **cheap strategy**, opting for **immediate beneficial moves** that **promise local improvements**.
- **No Backtracking:**
  - Unlike other algorithms, Hill Climbing Algorithm in AI **does not revisit or reconsider previous decisions**, persistently **moving forward** in the quest for the optimal solution.

Hill Climbing

- **Advantages**
  - Hill Climbing is a **simple** and **intuitive algorithm** that is easy to understand and implement.
  - It can be used in a **wide variety of optimization problems**, including those with a large search space and complex constraints.
  - Hill Climbing is often very **efficient** in **finding local optima**, making it a good choice for problems where a good solution is needed quickly.
  - The algorithm can be easily **modified** and **extended** to include **additional heuristics** or **constraints**.
- **Disadvantages**
  - Hill Climbing can **get stuck** in **local optima**, meaning that it may not find the global optimum of the problem.
  - The algorithm is **sensitive** to the **choice** of **initial solution**, and a poor initial solution may result in a poor final solution.
  - Hill Climbing **does not explore** the **search space** very **thoroughly**, which can limit its ability to find better solutions.
  - It may be **less effective** than **other optimization algorithms**, such as genetic algorithms or simulated annealing, for certain types of problems.

Simple Hill Climbing

- *Simplest way to implement a hill climbing algorithm.*
- **Evaluate the initial state.**
  - If it is a **goal state** then **stop** and **return success**.
  - Otherwise, make the **initial state** as the **current state**.
- **Loop** until the **solution state** is **found** or there are **no new operators present** which can be applied to the current state.
  - **Select a state** that has **not been yet applied** to the **current state** and **apply** it to **produce** a **new state**.
  - **Perform these to evaluate the new state.**
    - If the **current state** is a **goal state**, then **stop** and **return success**.
    - If it is **better** than the **current state**, then **make** it the **current state** and proceed further.
    - If it is **not better** than the **current state**, then **continue** in the **loop until** a **solution** is **found**.
- Exit from the function.

Features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Steepest Hill Climbing

- A variation of simple hill climbing algo.
- Algorithm **examines all the neighboring nodes** of the **current state** and **selects one neighbor node** which is **closest to the goal state**.

- **Step 1:** **Evaluate** the **initial state**,
  - if it is **goal state** then **return success** and stop,
  - **else** make **current state** as **initial state**.
- **Step 2:** **Loop until** a **solution** is **found** or the **current state** does **not change**.
- **Step 3:** Let **SUCC** be a state such that **any successor** of the **current state** will be **better** than **it**.
  - For each **operator** that **applies** to the **current state**:
- **Step 4:** **Apply** the **new operator** and **generate** a **new state**.
- **Evaluate** the **new state**.
  - If it is **goal state**, then **return it** and **quit**, else **compare** it to the **SUCC**.
  - If it is **better** than **SUCC**, then set **new state** as **SUCC**.
  - If the **SUCC** is **better** than the **current state**, then set **current state** to **SUCC**.
- **Step 5:** Exit.

Stochastic Hill Climbing

- It **does not examine all** the **neighboring nodes** before **deciding** which **node** to **select**.
- It just **selects** a **neighboring node** at **random** and **decides** (based on the amount of improvement in that neighbor) **whether** to **move** to **that neighbor** or to **examine another**.

- ***Algorithm***
- **Evaluate** the **initial state**.
  - If it is a ***goal state*** then ***stop*** and ***return success***.
  - ***Otherwise***, make the ***initial state*** the ***current state***.
- *Repeat* these *steps* until a *solution* is *found* or the *current state* does *not change*.
  - **Select** a **state** that **has not been** yet **applied** to the **current state**.
  - ***Apply*** the ***successor function*** to the ***current state*** and ***generate all*** the ***neighbor states***.
  - ***Among*** the ***generated neighbor states*** which are ***better*** than the ***current state*** choose a ***state randomly*** (or based on some probability function).
  - If the chosen state is the ***goal state***, then ***return success***, else make it the ***current state*** and ***repeat step*** 2 of the second point.
- Exit from the function.

Problems in Hill Searching

Local Maximum:

Is a peak state which **is better than each of its neighboring states**, but there is **another state** also **present** which is **higher** than the **local maximum**.

Solution: **Backtracking** technique can be a solution of the local maximum in state space landscape.

Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

Plateau:

Flat area of the **search space** in which all the **neighbor states** of the **current state contains** the **same value**, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to **take big steps or very little steps while searching**, to solve the problem. **Randomly select a state which is far away from the current state** so it is possible that the algorithm could find non-plateau region.

Ridges:

A **special form** of the **local maximum**. It has an **area** which is **higher** than its **surrounding areas**, but **itself has a slope**, and **cannot be reached in a single move**.

Solution: With the use of **bidirectional search**, or by moving in different directions, we can improve this problem.

Simulated annealing

- Hill-climbing algorithm which *never makes a move towards a lower value guaranteed to be **incomplete*** as it can get stuck on a ***local maximum***.
  - And if algorithm applies a **random walk**, by moving a successor, then it may **complete** but **not efficient**.

Simulated annealing

- **Simulated Annealing**: Offers both **efficiency** and **completeness**.
  - In mechanical terms, **Annealing** is a *process of hardening a metal or glass* to a *high temperature* then *cooling gradually*.
  - So this *allows* the *metal* to reach a *low-energy crystalline state*.
  - Same process is used in simulated annealing algo.
    - The **algorithm** picks a **random move**, **instead** of **picking** the **best move**.
    - If the **random move improves** the **state**, then it **follows the same path**.
    - Otherwise, the algo. **follows the path** which has a **probability of less than 1** or it **moves downhill** and **chooses another path**.

Simulated annealing

- Resembles the annealing process used to cool metals slowly to reach an ordered (low-energy) state
- Basic idea:
  - Allow "bad" moves occasionally, depending on "temperature"
  - High temperature => more bad moves allowed, *shake the system out of its local minimum*
  - Gradually reduce temperature according to some schedule

Simulated Annealing

- Consider gradient descent,
  - Getting a ping-pong ball into the deepest gap in a bumpy surface.
  - If we let the ball roll, it will come to rest at a local minimum.
  - If we shake surface, we can bounce the ball out of the local minimum.
- Simulated Annealing
  - Start by shaking hard and then gradually reduce the intensity of the shaking.
  - Quite similar to hill climbing.
  - Instead of picking the best move, however it picks a random move.
  - If the move improves the situation, it is always accepted.
  - Otherwise, the algorithm accepts the move with some probability less than 1

Local beam search

- Basic idea:
  - *K* copies of a local search algorithm, initialized randomly
  - For each iteration
    - Generate ALL successors from *K* current states
    - Choose best *K* of these to be the new current states

Or, K chosen randomly with a bias towards good ones

Beam search example (K=4)

Local beam search

- Why is this different from *K* local searches in parallel?
  - The searches ***communicate***! “Come over here, the grass is greener!”
- Differ from random restart search
  - Random restart runs independently.
- In local beam search, useful information is passed among the parallel search threads

Local beam search

- Lack of diversity among the k states.
- Stochastic beam search helps alleviate this problem
  - Choose *k* successors at *random*, with the *probability* of *choosing* a *given successor* being an *increasing function* of its *value*.
  - Resemble to the process of natural selection "Successors" (offspring) of a "*state*" (organism) populate the next generation according to its "*value*" (fitness)