**Informed Search Algorithms**

**Graphical user interface, text

Description automatically generated**

A strategy is defined by picking the order of node expansion

**heuristic function**

A strategy is defined by picking the order of node expansion

*A rule of thumb, simplification, or educated  guess that* reduces or limits the search for solutions in  domains that are difficult and poorly understood.”

h(n) = estimated cost of the cheapest path from node n to goal node.

If n is goal then h(n)=0

**Greedy best first search**

hSLD=straight-line distance heuristic.

In this example f(n)=h(n)

Expand node that is closest to goal

Evaluation

Completeness: Yes m – maximum depth of search space

Time complexity: = Space Complexity = Shape

Description automatically generated with medium confidence, optimality = no

Keeps all nodes in memory unlike DFS

**A\*** - -- Best-known form of best-first search.

Idea: avoid expanding paths that are assumed expensive.

Evaluation function f(n)=g(n) + h(n)

g(n) the cost (so far) to reach the node.

h(n) estimated cost to get from the node to the goal.

f(n) estimated total cost of path through n to goal.

**Recursive best-first search**

A\* search uses an admissible heuristic

A heuristic is admissible if it never overestimates the cost to reach the goal

Is optimistic Formally:

1. h(n) <= h\*(n) where h\*(n) is the true cost from n

2. h(n) >= 0 and h(G)=0 for any goal G.

e.g., hSLD(n) never overestimates the actual road distance

suboptimality

Suppose suboptimal goal G2 is in the queue.

Let n be an unexpanded (intermediate) node on a shortest path to optimal goal G.

f(G2 ) = g(G2 ) since h(G2 )=0

> g(G) since G2 is suboptimal

>= f(n) since h is admissible

Since f(G2) > f(n), A\* will never select G2 for expansion

Completeness: Yes

Time complexity: (exponential with path length)

Space complexity:(all nodes are stored)

Optimality: Yes

Cannot expand fi+1 until fi is finished.

A\* expands all nodes with f(n)< C\*

A\* expands some nodes with f(n)=C\*

A\* expands no nodes with f(n)>C\*

Also, it is optimally efficient – it does not expand more nodes when compared to other optimal algorithms such as Uniform Cost Search or Dijsktra.

Main idea: estimated heuristic costs ≤ actual costs

Admissibility: heuristic cost ≤ actual cost to goal

h(A) ≤ h\*(A) where h\* is the actual cost

Consistency: heuristic “arc” cost ≤ actual cost for each arc

h(A) – h(C) ≤ c(A,C)

or h(A) ≤ c(A,C) + h(C) (triangle inequality)

Consequences of consistency:

The f value along a path never decreases:

h(A) ≤ c(A,C) + h(C) => g(A) + h(A) ≤ g(A) + c(A,C) + h(C) – this means that f(A) ≤ f(C)

Therefore, the path taken by A\* graph search is optimal

Recursive Breadth first search

Text, letter

Description automatically generated

Recursive best-first search

Keeps track of the f-value of the best-alternative path available.

If current f-values exceeds this alternative f-value than backtrack to alternative path.

Upon backtracking change f-value to best f-value of its children.

Re-expansion then still possible.

Diagram

Description automatically generated

Unwind recursion and store best f-value for current best leaf Fagaras

result, f [best] ← RBFS(problem, best, min(f\_limit, alternative))

best is now Rimnicu Vilcea (again). Call RBFS for new best

Subtree is again expanded.

Best alternative subtree is now through Timisoara.

Solution is found since because 447 > 417.

RBFS is more efficient than A\*

Expands lesser nodes than A\*

Like A\*, optimal if h(n) is admissible

Space complexity is O(bd).

If d is the search tree depth at which the f-limit is reached, then RBFS will never expand more than bd nodes

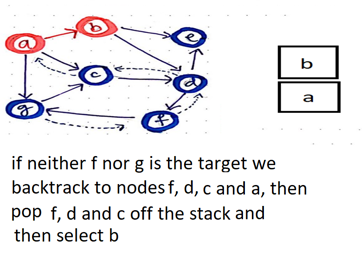
Time complexity difficult to characterize

Depends on accuracy if h(n) and how often best path changes.

DEPTH FIRST SEARCH

DFS is a search procedure that is applied to a tree or graph data structure that determines paths from a given start node S to a given target node T.

If T is not given, then it yields all paths between S and other nodes that can be reached from S.



A recursive implementation is a natural solution for DFS as the recursion will manage the stack and backtracking process automatically.

Given a graph G represented by a list of adjacentEdges(v) that contains all edges from vertex v, we have:

procedure DFS(G, v) is

    label v as discovered

    for all directed edges from v to w that are in G.adjacentEdges(v) do

        if vertex w is not labeled as discovered then

            recursively call DFS(G, w)

A recursive implementation is a natural solution for DFS as the recursion will manage the stack and backtracking process automatically.

Given a graph G represented by a list of adjacentEdges(v) that contains all edges from vertex v, we have:

procedure DFS(G, v) is

    label v as discovered

    for all directed edges from v to w that are in G.adjacentEdges(v) do

        if vertex w is not labeled as discovered then

            recursively call DFS(G, w)

**Non Recursive (iterative) Algorithm for DFS**

procedure DFS\_iterative(G, v) is

    let S be a stack

    S.push(iterator of G.adjacentEdges(v))

    while S is not empty do

        if S.peek().hasNext() then

            w = S.peek().next()

            if w is not labeled as discovered then

                label w as discovered

                S.push(iterator of G.adjacentEdges(w))

        else

            S.pop()

Time and Space Complexity

Asymptotic time complexity is O(V+E), where V is the number of vertices and E is the number of edge

In the worst case DFS will have to visit every node V and every edge E and therefore will take O(V+E) time

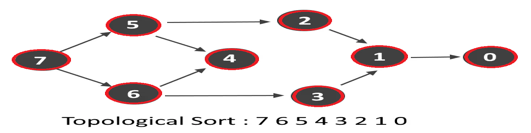
Asymptotic space complexity is O(V) as in the worst case the entire set of nodes needs to be stored on the stack

Applications of DFS

Finding components in a graph

This is an important problem with many real life applications such as transportation analysis, network partition analysis in routing systems, finding faults in electrical grids, VLSI design, etc.

It turns out that DFS can find the number of connected components with the help of a couple of simple modifications.



Topological sort on a directed graph is a linear ordering of its  vertices such that for every directed edge ab from vertex a to vertex b, a comes before b in the ordering.

Topological sort is required when we want to order vertices to satisfy some constraints such as in job scheduling – job B can only start after job A completes

Topological Sort

L ← empty list that will contain the sorted nodes

while nodes exist without a permanent mark do

    select an unmarked node n

    visit(n)

function visit(node n)

    if n has a permanent mark then

        return

    if n has a temporary mark then

        stop   (not a DAG)

    mark n with a temporary mark

    for each node m with an edge from n to m do

        visit(m)

    remove temporary mark from n

    mark n with a permanent mark

    add n to head of L

Real life application of dfs (MAZE example)

Finding a path in the maze that leads to the exit

At the first opening we can either go down or go right – each choice corresponds to a node

As always, edges corresponding to two connected nodes

When we hit a wall we backtrack and go backwards along the path to the previous decision point Dprev

Admissible Heuristic: A function is said to be admissible if it never overestimates

the cost of reaching the goal.

Local search= use single current state and moves to neighboring states.

Uses very little memory

Often finds reasonable solutions in large or infinite state spaces.

Are also useful for pure optimization problems.

Find best state according to some objective function.

e.g. survival of the fittest as a metaphor for optimization.

Hill climb Search

Is a process that continuously moves in the direction of increasing value

It terminates when a peak is reached.

Hill climbing does not look ahead of the immediate neighbors of the current state.

Hill-climbing chooses randomly among the set of best successors, if there is more than one.

Hill-climbing is a form of greedy local search

Text

Description automatically generated

8-queens problem (complete-state formulation).

Successor function: move a single queen to another square in the same column.

Heuristic function h(n): the number of pairs of queens that are attacking each other.

a) shows a state of h=17 and the h-value for each possible successor.

b) A local minimum in the 8-queens state space (h=1).

Min-conflicts: Chooses Randomly any variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints

example A two-step solution for an 8-queens problem using min-conflicts heuristic.

At each stage a queen is chosen for reassignment in its column.

The algorithm moves the queen to the min-conflict square breaking ties randomly.

Advantages of local search

The runtime of min-conflicts is roughly independent of problem size.

Solving the millions-queen problem in roughly 50 steps.

Local search can be used in an online setting.

Backtrack search requires more time

**Drawbacks**

Ridge = sequence of local maxima difficult for greedy algorithms to navigate

Plateau = an area of the state space where the evaluation function is flat.

Gets stuck 86% of the time.

**However**

Escaping shoulders: Sideways moves

If no uphill or downhill moves are possible then allow sideways moves in the hope that min conflict can escape from a local minimum

For 8 queens sideway moves up to a limit of 100 makes a drastic improvement in success rate from 14% to 94%

The improvement did come with a price:

It now took an average of 21 steps to work out a solution

Failures took 64 steps on the average

Hill-climbing variations

Stochastic hill-climbing

Random selection among the uphill moves.The selection probability can vary with the steepness of the uphill move.

First-choice hill-climbing

cf. stochastic hill climbing by generating successors randomly until a better one is found.

Random-restart hill-climbing

Tries to avoid getting stuck in local maxima by restarting at a random point in the state space

**Simulated annealing**

Escape local maxima by allowing “bad” moves.

Idea: but gradually decrease their size and frequency.

Origin; metallurgical annealing – heating metal to a very high temperature followed by cooling.

Bouncing ball analogy:

Shaking hard (= high temperature).

Shaking less (= lower the temperature).

If T decreases slowly enough, best state is reached.

Applied for VLSI layout, airline scheduling, etc.

Text

Description automatically generated

characteristics

SA is not an actual algorithm as it does not have exact specifications for the temperature schedule and how the next solution is to be obtained

SA is thus referred to as a metaheuristic algorithm

SA only becomes an algorithm when the schedule and next solution methods are specified for a given problem that we wish to solve

Unlike hill climbing it can move out of local maxima as it accepts solutions that are of lesser value than the current value

**Local beam search**

Keep track of k states instead of one

Initially: k random states

Next: determine all  successors of k states

If any of successors is goal → finished

Else select k best  from successors and repeat.

Major difference with random-restart search

Information is shared among k search threads.

Can suffer from lack of diversity.

Stochastic variant: choose k successors proportionally to state success.

Local beam search

A generalization of hill climbing algorithm

Start with P candidates

Keep the best p neighbors of these p candidates in the next round

Stochastic Local beam search

Similar to local beam search

Randomly choose k neighbors in the next round

Keep better neigbors , keep worst neighbors with probability

GENETIC ALGORITHM

Text

Description automatically generated with medium confidence

Variant of local beam search with sexual recombination.

Diagram

Description automatically generated

**Exploration problems**

Until now all algorithms were offline.

Offline= solution is determined before executing it.

Online = interleaving computation and action

Online search is necessary for dynamic and semi-dynamic environments

It is impossible to take into account all possible contingencies.

Used for exploration problems:

Unknown states and actions.

e.g., any robot in a new environment, a newborn baby,…

Until now all algorithms were offline.

Offline= solution is determined before executing it.

Online = interleaving computation and action

Online search is necessary for dynamic and semi-dynamic environments

It is impossible to take into account all possible contingencies.

Used for exploration problems:

Unknown states and actions.

e.g., any robot in a new environment, a newborn baby,…

online Search problem

* Agent knowledge:

ACTION(s): list of allowed actions in state s

C(s,a,s’): step-cost function      (After s’ is determined!)

GOAL-TEST(s)

An agent can recognize previous states.

Actions are deterministic.

Access to admissible heuristic h(s) e.g., Manhattan distance

**Online search agents**

The agent maintains a map of the environment.

Updated based on percept input.

This map is used to decide next action.

State space is safely explorable – in the sense that a goal state is always reachable.

Note difference with e.g. A\*

An online version can only expand the node it is physically in (local order)

Text

Description automatically generated