State Space Search

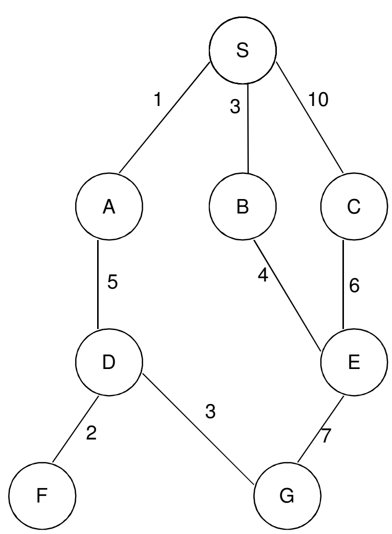
Uninformed/Blind Search

- **Breadth First Search**
- **Depth First Search**
- **Depth Limited Search**
- **Bidirectional Search**

Informed/Heuristic Search

- **Hill Climbing Search**
- **A* Algorithm**

General Graph search Algorithm



Graph G = (V,E)

1) Open List : S $^{(Ø, 0)}$
Closed list : Ø

2) OL : A$^{(S,1)}$, B$^{(S,3)}$, C$^{(S,10)}$
CL : S

3) OL : B$^{(S,3)}$, C$^{(S,10)}$, D$^{(A,6)}$
CL : S, A

4) OL : C$^{(S,10)}$, D$^{(A,6)}$, E$^{(B,7)}$
CL: S, A, B

5) OL : D$^{(A,6)}$, E$^{(B,7)}$
CL : S, A, B , C

6) OL : E$^{(B,7)}$, F$^{(D,8)}$, G$^{(D, 9)}$
CL : S, A, B, C, D

7) OL : F$^{(D,8)}$, G$^{(D,9)}$
CL : S, A, B, C, D, E

8) OL : G$^{(D,9)}$
CL : S, A, B, C, D, E, F

9) OL : Ø
CL : S, A, B, C, D, E, F, G

Steps of GGS

1. Create a search graph *G*, consisting only of the start node *S*; put *S* on a list called *OPEN.*

2. Create a list called *CLOSED* that is initially empty.

3. Loop: if *OPEN* is empty, exit with failure.

4. Select the first node on *OPEN*, remove from *OPEN* and put on *CLOSED*, call this node *n*.

5. if *n* is the goal node, exit with the solution obtained by tracing a path along the pointers from *n* to *s* in *G*.

6. Expand node *n*, generating the set *M* of its successors that are not ancestors of *n*.

GGS steps (contd.)

7. Establish a pointer to *n* from those members of *M* that were not already in *G* (*i.e.*, not already on either *OPEN* or *CLOSED*). Add these members of *M* to *OPEN*. For each member of *M* that was already on *OPEN* or *CLOSED*, decide whether or not to redirect its pointer to *n*. For each member of M already on *CLOSED*, decide for each of its descendents in *G* whether or not to redirect its pointer.
8. Reorder the list *OPEN* using some strategy.
9. Go *LOOP*.

Measuring problem-Solving performance

What makes one search scheme better than another?

Completeness: Guarantee to find a solution?

Time complexity: How long is it to find a sol.?

Optimality: Does the srategy find the shortest path?

Space complexity: How much memory is needed?

Branching Factor b: maximun number of sucessors of any node

Breadth First Search

- Simple Strategy
- The root is expanded first, Then all its successors, Then all their successors
- At a given depth, All nodes are expanded.
- With branching factor b, at level d, we have

$b+b^2+b^3+...b^d = O(b^d)$ Nodes

- At level 12 with branching factor 10, we 10^{13} nodes
- Space Problem !

BDF

Completeness?

Yes, if solution exists, there is a guaratee to find it

Time complexity?

$O(b^d)$

Space complexity?

$O(b^d)$

Optimality?

yes

Bidirectional Search

BFS in both directions

How could this help?

b^L vs $2b^{L/2}$

- Can reduce time complexity,
- Not always applicable
- May require lots of space
- Hard to implement

BDF

Completeness?

Yes, if solution exists, there is a guarantee to find it

Time complexity?

$O(b^d)$, b is branching factor, d is least cost to goal

Space complexity?

$O(b^d)$

Optimality?

yes

Depth First Search

- Always expand deepest node in the fringe of the tree.
- Modest memory requirement, stores only single path from root to leaf.
- With branching factor b, at level d, we store only bm+1 i.e. O(bm)
- It may stuck in an infinite path and never finds solution

DFS

Completeness?

Yes, assuming state space finite. If the space is not finite, then no guarantee

Time complexity?

O(m), can do well if lots of goals

Space complexity?

O(n), n deepest point of search

Optimality?

No may find a solution with long path

Depth-limited Search

Put a limit to the level of the tree

DFS, only expand nodes depth ≤ L.

Completeness?

No, if L ≤ d.

Time complexity?

$O(b^L)$

Space complexity?

O(L)

Optimality?

No

Iterative Deepening

- Calls depth-limited search with increasing limits until goal is found

  Completeness?

  Yes.

  Time complexity?

  $O(b^d)$

  Space complexity?

  O(d)

  Optimality?

  Yes

Remarks

- BFS works as a queue. Pick the leftmost element of the open list , evaluate it and add its children to the end of the list, FIFO
- DFS works as a stack. Pick the leftmost element of the open list , evaluate it and add its children to the beginning of the list, LIFO

Informed Search

- Uses some knowledge -not from the definition of the problem -
- Can find solutions more efficient than uninformed
- Gerneral approach is *best-first-seach*
- A node is selected based on an *evaluation function f(n)*
- A node that **seems** to be best is picked and it may not be the actual best

Best First Search

- The Idea:
  - use an *evaluation function* for each node... estimate of ``desirability''
  - Expand most desirable unexpanded node

Implementation

- Fringe: is a queue sorted in decreasing order of desirability

Special cases

Gready

A*

Cost function $f(n)$

- A function f is maintained for each node

$f(n) = g(n) + h(n)$, n is the node in the open list

- "Node chosen" for expansion is the one with least f value
- $g(n)$ is the cost from root S to node n
- $h(n)$ is the estimated cost from node n to a goal
- For BFS: $f = 0$,
- For DFS: $f = 0$,
- For greedy g =0

Greedy search

- Expands a node it sees closest to a the goal
- $f(n) = h(n)$
- Resembles DFS in that it prefers to follow a single path all the way to the goal
- Also suffers from the same defects of DFS, it may stuck in a loop i.e. not complete As well as it is not optimal.

Hill climbing

This is a *greedy* algorithm
Expands a node it sees closest to a goal
$f(n) = h(n)$

The algorithm

select a heuristic function;

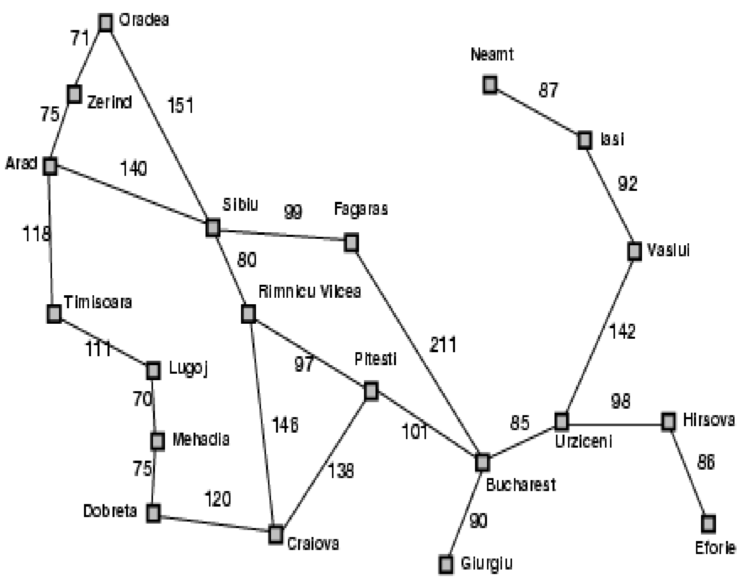
set C, the current node, to the highest-valued initial node;

Loop until success or no more children(fail)

select N, the highest-value child of C;

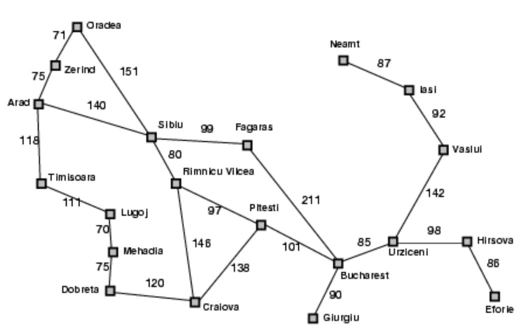
return C if its value is better than the value of N;

Hill Climbing search example



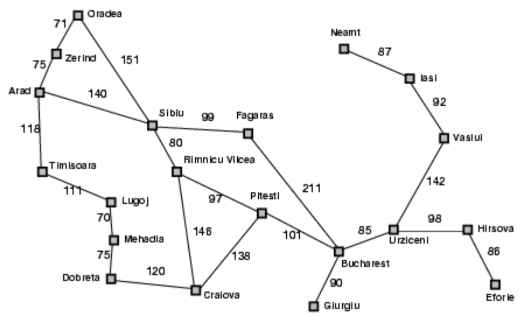
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Hill Climbing search example



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

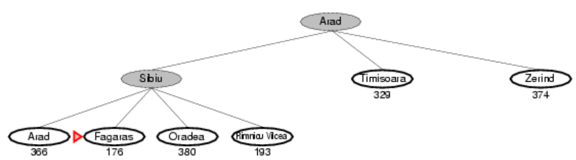
Hill Climbing search example

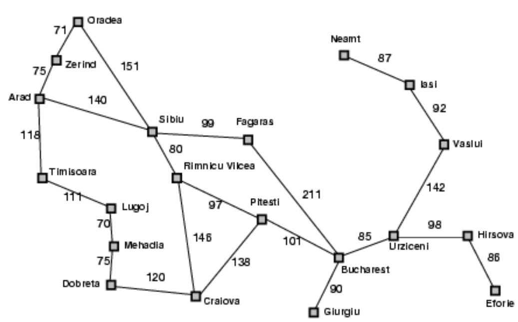


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Hill Climbing search example

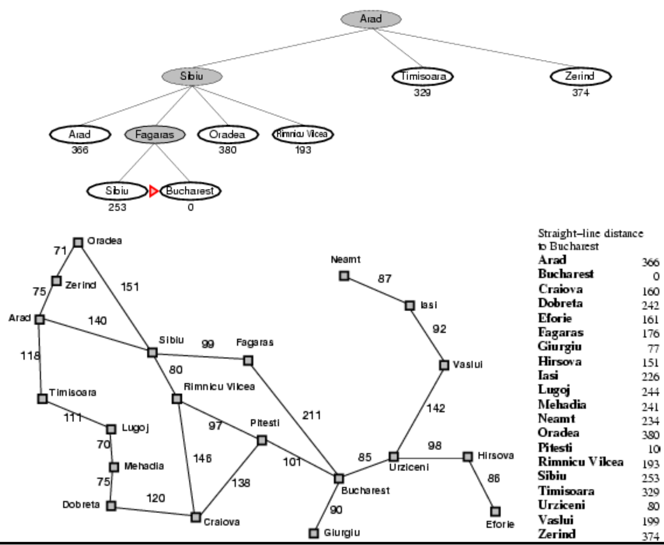




Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Hill Climbing search example



Hill climbing

<u>Complete</u>: No, Can get stuck in loop. Complete if loops are avoided.

<u>Time complexity</u>? $O(b^m)$, but with some good heuristic, it could give better results

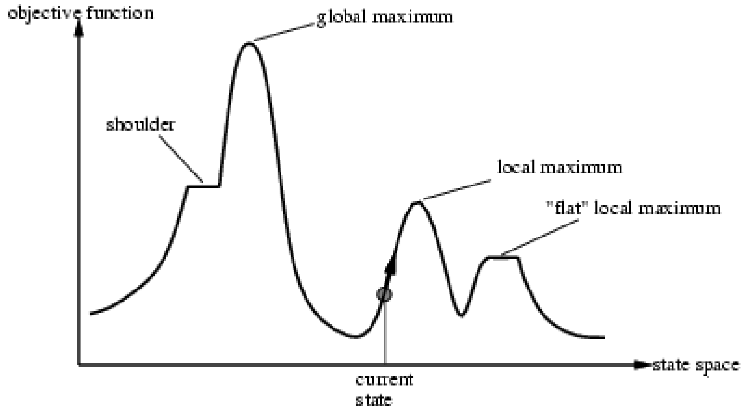
<u>Space complexity</u>? $O(b^m)$, keeps all nodes in memory

<u>Optimality</u>? No

e.g. Arad→Sibiu→Rimnicu Virea→Pitesti→Bucharest is shorter!

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima,…etc



Problems with hill climbing

1. Local maximum problem: there is a peak, but it is lower than the highest peak in the whole space.
2. The plateau problem: all local moves are equally unpromising, and all peaks seem far away.
3. The ridge problem: almost every move takes us down.

Problems with hill climbing

1. Local maximum problem: there is a peak, but it is lower than the highest peak in the whole space.
2. The plateau problem: all local moves are equally unpromising, and all peaks seem far away.
3. The ridge problem: almost every move takes us down.

Solution:

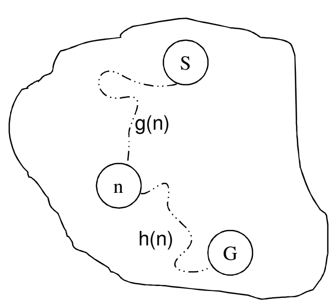
Random-restart hill climbing is a series of hill-climbing searches with a randomly selected start node whenever the current search gets stuck.

Algorithm A*

- One of the most important advances in AI search algs.
- Idea: avoid expanding paths that are already expensive

$$f(n) = g(n) + h(n)$$

- $g(n)$ = least cost path to n from S found so far
- $h(n)$ = estimated cost to goal from n
- $f(n)$ = estimated total cost of path through n to goal



The A* procedure

Hill-climbing (and its improved versions) may miss an optimal solution. Here is a search method that ensures **optimality** of the solution.

The algorithm

keep a list of partial paths (initially root to root, length 0);

repeat

succeed if the first path P reaches the goal node;

otherwise remove path P from the list;

extend P in all possible ways, add new paths to the list;

sort the list by the sum of two values: the real cost of P till now, and an estimate of the remaining distance;

prune the list by leaving only the shortest path for each node reached so far;

until

success or the list of paths becomes empty;

The A* procedure

A heuristic that never overestimates is also called **optimistic** or **admissible**.

We consider three functions with values ≥ 0:

- g(n) is the actual cost of reaching node n,
- h(n) is the actual *unknown* remaining cost,
- h*(n) is the optimistic estimate of h(n).

Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*,

  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Theorem: If *h(n)* is admissible, A^* using is optimal

Read the proof of the optimality of the goal node found by A