Intelligent Systems: Reasoning and Recognition

James L. Crowley

ENSIMAG 2 and MoSIG M1 Winter Semester 2013
Lecture 4 20 February 2013

Problem Solving as Planning
Heuristic Search with GRAPHSEARCH

Planning as Graph Search

A problem is defined by
a universe, {U},
an initial state, i
A set of Goal states, {G}.

Planning is the generation of a sequence of actions to transform i to a state $g \in \{G\}$

The "paradigm" for planning is "Generate and Test".

Given a current state, s
1) Generate all neighbor states {N} reachable via 1 action.
2) For each $n \in \{N\}$ test if $n \in \{G\}$. If yes, exit
3) Else Add {N} to a set {O} of open states to be explored
3) Select a next state, $s \in \{O\}$, extract s from {O} and loop.

Planning requires search over a graph for a path.

A taxonomy of graph search algorithms includes the following

1) Depth first search
2) Breadth first search
3) Heuristic Search (A or A* search)
4) Hierarchical Search

The first three are unified within the GRAPHSEARCH algorithm of Nilsson.

Graph searching has exponential algorithm complexity.
"knowledge" can be used to reduce the complexity.

Under certain conditions, heuristic search can be said to be "optimum". In this case it is said to be A*

Nilsson's GRAPHSEARCH algorithm

Breadth first, depth first and heuristic search are all variations on the same 3 step GRAPHSEARCH algorithm, depending on whether the Open list is a stack, queue or sorted.

The algorithm requires maintaining a list of "previously visited" states {C} (Closed list) and a list of available states to explore {O} (Open list).
While exploring the graph, Graphsearch constructs a search tree T.

Symbols :
- T : Search Tree
- {G} : Set of Goal States
- i : Departure State
- {N} : List of Neighbor States
- {O} : List of Open States
- {C} : List of Closed States
- n, s: Nodes representing states

GRAPHSEARCH Algorithm: Given a start state i and a set of Goal states {G}.

1) Create T, {O} and {C}, (initially empty).
2) Place i in {O}, and as root to T.
LOOP:
3) Extract s from {O}, add s to {C}:
    {O} <- {O} - s;
    {C} <- {C} + s;
4) If s $\in$ {G}, then EXIT with Success
    Construct a Solution stack with states from s to i.
    Un-stack the solution stack. This is the best path.
5) {N} <- Neighbors(s) /* {N} gets neighbors of s */
6) For each n $\in$ {N} IF n $\notin$ {C} THEN
    T <- T + child(n, s) /* add n to search tree as child of s */
    a) For dept first search treat {O} as a stack:
        {O}<- push(n, {O})
    b) For breadth first search treat {O} as a queue:
        {O}<- append({O}, n)
    c) For A* Calculate cost f(n) of path from i to n (g(n)) and from n to G (h(n)
        f(n) = g(n) + h(n)
        add (n, f(n)) to {O};
        sort {O} based on f(n)
7) if {O} $\neq$ {} (empty set) then Go to step 3 else EXIT with failure

Initially {C} and {O} are empty

Step 3 determines the nature of the search.

if {O} is a queue (FIFO) then the search is **Breadth First**
if {O} is a stack (LIFO) then the search is **Depth First**
if {O} is sorted based on a cost, f then the Search is Heuristic (or **Best First)**

Algorithmic Complexity

We estimate algorithm complexity with the Order operator O().
Algorithm complexity order is equivalent for all linear functions.

$O(AN+B) = O(N)$

The algorithm complexity of graph search depends on

b: The branching factor; The average number of neighboring states {N}
$b = E\{card(\{N\})\}$ (E{} is expectation)

d: Depth. The minimum number of nodes from i to {G}.

Breadth First search: {O} is a queue (FIFO)

For breadth first search, finding the optimal path requires exhaustive search.
Computation Cost $O(b^d)$, memory $O(b^d)$.

Depth First search: {O} is a stack (LIFO).

For depth first search, finding the optimal path requires exhaustive search, however
Computation Cost $O(b^d)$, memory O(d).
However, depth first requires setting a maximum depth d_{max}.

Heuristic search: {O} is sorted based on a cost, f.

For Heuristic search, we reduce the order by reducing the branching factor:
This give Computation and memory of $O(c^d)$ where $c \leq b$.

Heuristic Search is NOT exhaustive. We avoid unnecessary branches.

Nilssons GRAPHSEARCH provides Heuristic search in two forms.

Algorithm A : uses an arbitrary cost estimate.
Algorithm A* : uses an "optimal" cost estimate to produce "optimal" search.

Cost of a path through a state s, f(s), includes cost from inital to the current state (g(s), plus estimated cost from current state to a goal state, h(s).

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

A^* requires that the cost function and cost estimate meet the "optimality conditions".

Cost and Optimality of GRAPHSEARCH

A key problem is defining a notion of numerical "cost" for executing an action.
Cost can be any numerical value, but must obtain certain conditions for GRAPHSEARCH to be optimal.

Examples of cost: Euros, distance, time, risk, number of actions.

Notation :

i : initial state
g : goal state $g \in \{G\}$
$k(s_i, s_j)$: minimal theoretical cost between states s_i and s_j
$g^*(s) = k(i, s)$: The cost of the shortest path from i to s.
$h^*(s) = k(s, g)$: The cost of the shortest path from s to $g \in \{G\}$.

$f^*(s) = g^*(s) + h^*(s)$ The cost of the shortest path from i to g passing by s.

Problem: If we do not know the shortest path, how can we know $g^*(s)$ or $h^*(s)$?
Solution estimate the costs.
Define:

g(s) : estimated cost from i et s.
h(s) : estimated cost from s to g
$f(s) = g(s) + h(s)$

Nilsson showed that whenever $f(s) \le f^*(s)$, the first path that is found from s_1 to s_2 will always be the shortest.

This requires two conditions:

Condition 1: that the heuristic UNDER-ESTIMATES the cost.

$h(s) \leq h^*(s)$

Condition 2: that h(s) is a "monotonic" function. That is :

$$h(s_i) - h(s_j) \leq k(s_i, s_j)$$

This is almost always true whenever $h(s) \leq h^*(s)$!!

From this he showed that because first path from i to s is the shortest path and thus $g(s) = g^*(s)$!

Thus as long as $h(s) \leq h^*(s)$ then $f(s) \leq f^*(s)$ and the search is "optimal".

Nilsson called this the A* condition.
A* is "optimal" because the first path found is the shortest path.

Whenever the cost metric is the length of the path, then Euclidean distance to the goal provides an "optimal" heuristic!

This is also true for scalar multiples of distance, for example, time traveled or risk.

(assuming constant speed, distance = speed/time.)

Note that for $h(s) = 0$, h(s) meets the optimality condition because $h(s) \leq h^*(s)$!!
This is dijkstra's algorithm, used for network routing.

We can speed up the search by using a better h(s) that 0! However, the first solution found is always the best solution.

Example: Mobile Robot Path Planning

Domain knowledge can be used to guide this search.

To illustrate this, consider the problem of path planning for a mobile robot.

Navigation requires a "map". The classic map for path planning is a "network of places".

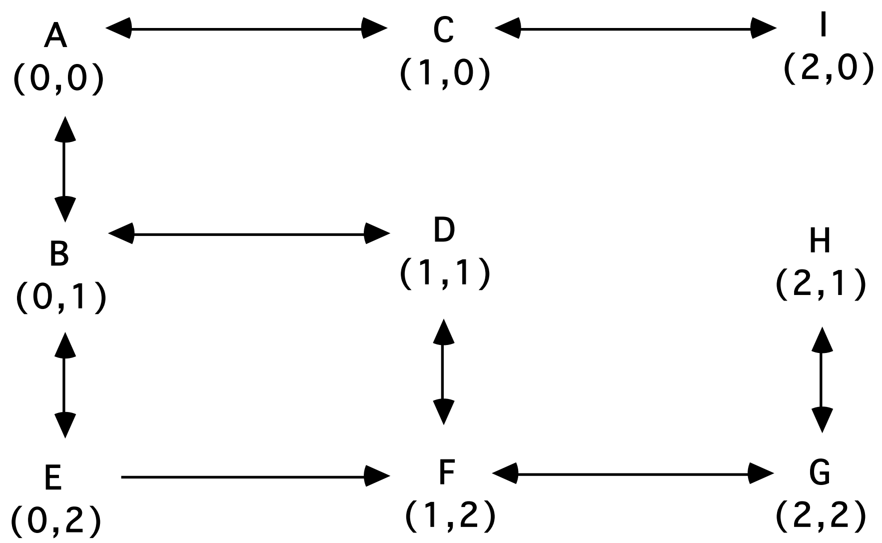
A place is defined as
1) A name
2) An inclusion test (a predicate At(x))
3) A list of "adjacent" places that can be reached by a single action.

The set of places compose a network. ("A network of places").

Navigation planning requires finding a sequence of places that lead from i to {G}.
The search for a path generates a tree of possible paths.

There are 3 forms of search algorithms that can be used to generate this tree:
Depth first, breadth first, and heuristic.

Let us use the following graph to illustrate these three algorithms.
Note that the places are labeled with coordinates (x,y) on a Cartesian map.
This allows us to define a metric to evaluate the "cost" of alternative paths.



Given that the robot is at place "E" and we require it to go to H.