Knowledge-based systems in Bioinformatics, 1MB602, 2006

Lecture 6: Search

Lecture overview

- Goal based agents
- Search terminology
- Specifying a search problem
- Search considerations
- Uninformed search
- Heuristic methods
- Iterative improvement algorithms

Goal based agents

- Intelligent agents act in such a way that the environment goes through a sequence of states that maximizes the performance measure
- The agent adopts a goal and aims to satisfy it
- The agent can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best one
- The process of looking for such a sequence is called search

Search terminology

- State
  - A situation the search can visit
- State space
  - The set of all states reachable from the initial state by any sequence of actions
- Path
  - A sequence of actions in the state space leading from one state to another
- Solution
  - A state with a particular property, i.e. solves the problem (achieves the task)
  - May be more than one solution to a problem
- Strategy
  - How to choose the next step in the path at any given state

Specifying a search problem

- Initial state
  - The start state of the search which holds description about the current state of the world
- Operators
  - Functions taking the search from one state to another
  - Specify how the agent can move around the search space
- Goal test
  - Has the search succeeded?

Route planning example

- Initial state
  - City the journey starts in
- Operators
  - Moving from one city to another
- Goal test
  - Located in a certain city?

2D folding example

- Initial state
  - A non-folded chain of amino acids (hydrophobic or hydrophilic identifiers)
- Operators
  - Different types of rotation
- Goal test
  - Protein completely folded?
  - Energy of protein below a certain predefined threshold

Search considerations

- What are we looking for?
  - What is interesting, the solution or the path to the solution?
- Completeness
  - Is the search guaranteed to find a solution when it exists?
  - Pruning vs. exhaustive searches
- Time and space complexity
- Optimality
  - Does the strategy find the highest-quality solution when there are several different solutions?
- Soundness
  - Is the path (and each step in the path) truth-preserving?
  - Important for automated reasoning
- Additional information
  - Uninformed search uses no additional information
  - Heuristic search take advantages of various information

Graph analogy

- A state space can be seen as a graph $G = (V,E)$, where V is a set of vertices (nodes) and E is a set of edges
- In a state space graph, V corresponds to the state space and E corresponds to transitions in G taking the search from one state (vertex) to another
- Choices that determines which vertex to expand and which edge to go down define the search strategy

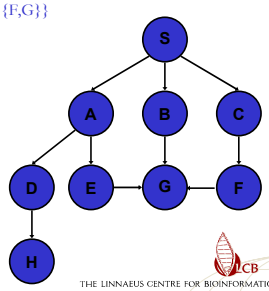
Formulizing search in a state space

$V = \{A, B, C, D, E, F, G, H, S\}$

$E = \{\{S,A\}, \{S,B\}, \{S,C\}, \{A,D\}, \{A,E\}, \{B,G\}, \{C,F\}, \{D,H\}, \{E,G\}, \{F,G\}\}$

The size of a problem is usually described in terms of the number of possible states

- Tic-tac-toe: 3^9 states
- Rubik's cube: $\approx 10^{19}$ states
- Chess: $\approx 10^{123}$ states



Formulizing search in a state space cont.

- Each vertex has a set of successor vertices
  - The set of all legal actions for each vertex
  - Each successor is determined by applying the edge's action to the predecessor vertex
- Expanding a vertex generates the successors of that vertex and adds them to the search graph
- The search algorithm maintains a list of successor vertices, a state list
  - The first item in the state list corresponds to the next state to expand
  - Expanding a vertex manipulates the state list by removing the visited vertex and adding the successor vertices to the state list
  - How to add depends on the search strategy

Search techniques

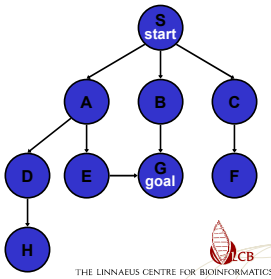
- Partial vs. complete search
  - Partial search: only interested in a solution, i.e. only a subset of the search space may be considered
  - Complete search: the best solution is desired, e.g. optimization problems
- Uninformed vs. informed search
  - Uninformed search: only the information of the current state and possible one-step transitions to other states are considered when choosing the next state in the path
  - Informed search: estimate path and/or action cost and use this information to choose the next vertex to expand

Breadth first search

The state list corresponds to a queue (first in first out)
The algorithm visits the vertices one level at a time

Current	State list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}

States visited: 7
States expanded: 6

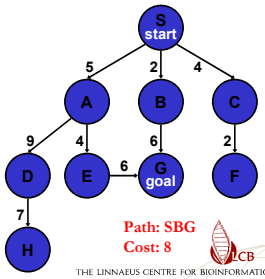


Associating costs with actions

- Often we are interested in the search path and the cost of that path instead of just the solution
- BFS again:

Current	State list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}

States visited: 7
States expanded: 6



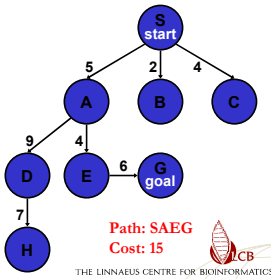
Path: SBG
Cost: 8

Depth first search

- The state list corresponds to a stack (last in first out)
- Always expands one of the nodes at the deepest level of the tree

Current	State list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G	{B,C}

States visited: 6
States expanded: 5



Path: SAEG
Cost: 15

BFS vs. DFS

- BFS
  - Complete
  - Optimal when all actions, i.e. edges, have the same cost
    - finds the shallowest goal state
  - Time complexity: $O(b^d)$
    - b: branching factor, d: solution depth
  - Space complexity: $O(b^d)$
  - Memory requirements are a bigger problem than the execution time
- DFS
  - Can get stuck going down the wrong path
    - Not complete (may go down a path of infinite depth)
    - Not optimal
    - Should be avoided for search graphs with large or infinite maximum depths
  - Time complexity: $O(b^m)$
    - b: branching factor, m: maximum depth of the search graph
  - Space complexity: O(bm) (need only store the nodes in one path)

Other uninformed search strategies

- UCS: uniform cost search (branch and bound)
  - Use priority queue for state list, sorted by path cost
  - g(v) = cost of path from start vertex s to current vertex v
  - Sort vertices by increasing value of g
  - Complete, optimal, time and space complexity: $O(b^d)$
  - Note: the same as BFS if g(v) = DEPTH(v)!
- IDS: iterative deepening search
  - Do DFS to depth 0, then depth 1, then dept 2, etc until a solution is found
  - Combines the best of BFS and DFS
  - Complete, optimal, space complexity: O(bd), time complexity: $O(b^d)$
- Bidirectional search
  - Simultanously search both forward from the initial state and backward from the inital state and backwards from the goal

Informed search

- In informed search strategies we introduce an evaluation function that associates a value to each vertex describing the desire to expand that vertex
- The state list is order so that the vertex with the best evaluation is expanded first → best first search
- Evaluation of a vertex = estimation of path cost from vertex to solution
- The evaluation function is often called a heuristic function h:
  - h(v) = estimated cost of the cheapest path from the state at vertex v to goal state
  - Heureka ("I have found it", Archimedes)
  - Problem specific

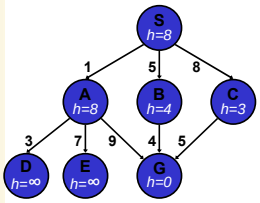
Best first search strategies

- Greedy search (Best-first search)
  - Minimize estimated cost to reach a goal
  - Evaluation function: $h(v)$ = estimated cost to the goal from v
  - The vertex whose state is judged to be closest to the goal is always expanded first (best first)
  - Neither optimal nor complete

Best first search strategies cont.

- A* search
  - Evaluation function $f(v) = g(v) + h(v)$ = estimated cost of the cheapest solution through v
    - $h(v)$ = estimated cost to the goal from v
    - $g(v)$ = the cost of the path so far through v
  - Important restriction: h must be admissible, i.e. never overestimate the cost to reach the goal – optimistic thinking
    - $h(v) \leq h^*(v)$ always holds, where $h^*(v)$ is the actual cost
  - Complete, optimal, and optimally efficient among all optimal search algorithms
  - Still exponential time (and memory) unless $|h(v) - h^*(v)| < O(\log h^*(v))$
    - Unfortunatly, this is seldom the case
  - Note: with $h(v) = 0$, this is a UCS

Determining f(v)



v	g(v)	h(v)	f(v)	h*(v)
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	∞	∞	∞
E	8	∞	∞	∞
G	10/9/13	0	10/9/13	0

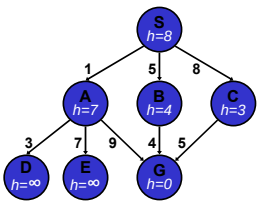
$g(v)$ = actual cost to get to vertex v from start
$h(v)$ = estimated cost to get to a goal from vertex v
$f(v) = g(v) + h(v)$
actual cost to get from start to v plus estimated cost from v to goal

Optimal path = SBG
Cost = 9
Admissible!

A* search example

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

Current	State list
	{S:0+8}
S	{A:1+7,B:5+4,C:8+3}
A	{B:9,G:1+9+0,C:11, D:1+3+∞,E:1+7+∞}
B	{G:5+4+0,G:10,C:11, D:∞,E:∞}
G	{C:11,D:∞,E:∞}



Path: S,B,G
Cost: 9

Heuristics

- Heuristic methods rely on good heuristics
- Should give the best score to the state we are looking for, and good scores to similar states
- Should be easy and quick to compute
- Better heuristics (less optimistic) translate directly into higher efficiency in time and memory!
- Some examples of heuristics
  - route planning: staight line distance
  - protein folding: physical, free energy
  - differences between observed and expected data
  - cost, profit (economic applications)
  - cost, production time, utilization of resources (industrial applications)

A variant of A* search

- Reducing memory requirements by turning A* into IDA*, Iterative Deepening A*
- Modify IDS to use an f-cost limit rather than a depth limit
- Each iteration expands all vertices inside the contour for the current f-cost
- Once the search inside a given contour has been completed, a new iteration is started using a new f-cost for the new contour
- Complete and optimal
- Memory requirments $O(bm)$: depth first search
- In the worst case, every node in the state space has a different f-cost, thus if A* examines k nodes then IDA* will examine $O(k^2)$ nodes

Memory-efficient A*: SMA* search

- IDA* repeat states in and between iteration
- SMA* uses a queue to store visited nodes sorted by f-cost
- Prune the worst-case node from memory when space is needed for a better one
- Uses all available memory: specify a maximum number of nodes MAX that can be stored
- Guaranteed to be optimal if MAX is at least as large as the number of nodes in the optimal solution

Iterative improvements algorithms

- In optimization problems the desired solution is the maximal (minimal) value of a function
  - Problem example: maximize the number of hydrophobic neighbors by two dimensional folding
- Applicable if each state contain all the information required for a solution (i.e. the path is irrelevant)
- Iterative improvement strategy
  - Start with a complete configuration and make modifications to improve its quality
- Two types of iterative improvement algorithms
  - Hill climbing
    - Choose as the next state to expand the state with the highest value (if higher than the current)
  - Simulated annealing
    - Allow temporary changes that makes things worse, to avoid getting stuck in local maxima

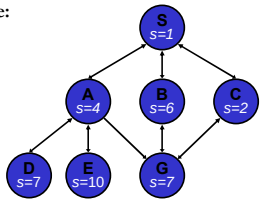
Local search optimization

- In optimization applications we want to find the best state, i.e. the state with the highest score.
- To be sure to find the best state, all states have to be examined.
- In many real-world applications the number of states is huge, so there is no way to do an exhaustive search.
- In these cases there are algorithms to find solutions that are reasonably good, although they might not be optimal.

→ local search optimization

Local search optimization

- Assumption: States that are close to each other have similar scores.
- Basic idea: Look through states in the neighborhood and gradually move to better and better states.
- Example:



Local search optimization cont.

- Problems:
  - Local search methods can get stuck in local optima → no guarantee to find the optimal solution.
  - Usually no way to know how much the obtained result differs from the global optimum.
  - Obtained result depends on starting state.
- Ways to deal with this:
  a) Restart the search with different (random) start states
  b) Occasionally make sub-optimal moves (simulated annealing)
  c) Force the search to explore different parts of the search space (tabu search)

Hill climbing

Hill climbing is the basic form of local search (example shown before)

Simulated annealing

- Basic idea: Sometimes make random sub-optimal moves. Sub-optimal moves are more common in the beginning of the search
- When should the function make random steps?
  - Depends on the temperature, which depends on the time.
  - Temperature decreases throughout the search.
- This requires quite a few parameter settings

Tabu search

- Also based on hill climbing, but some moves become illegal (hence the name Tabu)
- We have to choose among the allowed neighbor (i.e. that are not on the Tabu list)
- If a neighbor is much better than the best state, we may still use it even if it is on the Tabu list (aspiration criterion)
- Which states are on the Tabu list
  - Allready visited states
  - States with certain properties
  - ...

References

- S. Russell, P. Norvig, Artificial intelligence: a modern approach, Prentice-Hall, Upper Saddle River, New Jersey, 1995
- Z. Michalewicz, D.B. Fogel, How to Solve It: Modern Heuristics, Springer, 2000
- S. Russell, Efficient memory-bounded search methods, ECAI 92: 10th European Conference on Artificial Intelligence Proceedings, 1992
- E. Keedwell, A. Narayanan, Intelligent bioinformatics: the application of artificial intelligence techniques to bioinformatics problems. Chichester : John Wiley, cop. 2005

LCB
THE LINNAEUS CENTRE FOR BIOINFORMATICS
http://www.lcb.uu.se