Artificial intelligence 1: informed search

Lecturer: Tom Lenaerts
SWITCH, Vlaams Interuniversitair Instituut voor Biotechnologie

Outline

Informed = use problem-specific knowledge
Which search strategies?
– Best-first search and its variants
Heuristic functions?
– How to invent them
Local search and optimization
– Hill climbing, local beam search, genetic algorithms,
Local search in continuous spaces
Online search agents

Previously: tree-search

function TREE-SEARCH(problem, fringe) return a solution or failure
fringe ← INSERT(MAKE-NODE(INITIAL-STATE(problem)), fringe)
loop do
  if EMPTY?(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST(problem) applied to STATE(node) succeeds
     then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
end loop

A strategy is defined by picking the order of node expansion

Best-first search

General approach of informed search:
– Best-first search: node is selected for expansion based on an evaluation function f(n)
Idea: evaluation function measures distance to the goal.
– Choose node which appears best
Implementation:
– fringe is queue sorted in decreasing order of desirability.
– Special cases: greedy search, A* search

A heuristic function

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

More information later.

Romania with step costs in km

hSDL = straight-line distance heuristic.
hSDL can NOT be computed from the problem description itself
In this example f(n)=h(n)
– Expand node that is closest to goal
= Greedy best-first search
Assume that we want to use greedy search to solve the problem of travelling from Arad to Bucharest.
The initial state=Arad

The first expansion step produces:  
- Sibiu, Timisoara and Zerind  
Greedy best-first will select Sibiu.

If Sibiu is expanded we get:  
- Arad, Fagaras, Oradea and Rimnicu Vilcea  
Greedy best-first search will select: Fagaras

If Fagaras is expanded we get:  
- Sibiu and Bucharest  
Goal reached !!  
- Yet not optimal (see Arad, Sibiu, Rimnicu Vilcea, Pitesti)

Completeness: NO (cfr. DF-search) 
- Check on repeated states 
- Minimizing h(n) can result in false starts, e.g. Iasi to Fagaras.

Time complexity?  
- Cfr. Worst-case DF-search $O(b^m)$  
(with m is maximum depth of search space) 
- Good heuristic can give dramatic improvement.
Greedy search, evaluation

Completeness: NO (cfr. DF-search)
Time complexity: $O(b^n)$
Space complexity: $O(b^m)$
  - Keeps all nodes in memory

---

A* search

Best-known form of best-first search.
Idea: avoid expanding paths that are already 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.

---

A* search

A* search uses an admissible heuristic
  - A heuristic is admissible if it never overestimates the cost to reach the goal
  - Are optimistic

Formally:
1. $h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from $n$
2. $h(n) \geq 0$ so $h(G) = 0$ for any goal $G$.

E.g. $h_{SLD}(n)$ never overestimates the actual road distance

---

Romania example

Find Bucharest starting at Arad
  - $f(\text{Arad}) = c(?,\text{Arad}) + h(\text{Arad}) = 0 + 366 = 366$
A* search example

Expand Arrad and determine $f(n)$ for each node:
- $f(\text{Sibiu}) = c(\text{Arad, Sibiu}) + h(\text{Sibiu}) = 140 + 253 = 393$
- $f(\text{Timisoara}) = c(\text{Arad, Timisoara}) + h(\text{Timisoara}) = 118 + 329 = 447$
- $f(\text{Zerind}) = c(\text{Arad, Zerind}) + h(\text{Zerind}) = 75 + 374 = 449$
Best choice is Sibiu

A* search example

Expand Sibiu and determine $f(n)$ for each node:
- $f(\text{Arad}) = c(\text{Sibiu, Arad}) + h(\text{Arad}) = 280 + 366 = 646$
- $f(\text{Fagaras}) = c(\text{Sibiu, Fagaras}) + h(\text{Fagaras}) = 239 + 179 = 415$
- $f(\text{Oradea}) = c(\text{Sibiu, Oradea}) + h(\text{Oradea}) = 291 + 380 = 671$
- $f(\text{Rimnicu Vilcea}) = c(\text{Sibiu, Rimnicu Vilcea}) + h(\text{Rimnicu Vilcea}) = 220 + 192 = 413$
Best choice is Rimnicu Vilcea

A* search example

Expand Rimnicu Vilcea and determine $f(n)$ for each node:
- $f(\text{Craiova}) = c(\text{Rimnicu Vilcea, Craiova}) + h(\text{Craiova}) = 360 + 160 = 526$
- $f(\text{Pitesti}) = c(\text{Rimnicu Vilcea, Pitesti}) + h(\text{Pitesti}) = 317 + 100 = 417$
- $f(\text{Sibiu}) = c(\text{Rimnicu Vilcea, Sibiu}) + h(\text{Sibiu}) = 300 + 253 = 553$
Best choice is Fagaras

A* search example

Expand Fagaras and determine $f(n)$ for each node:
- $f(\text{Sibiu}) = c(\text{Fagaras, Sibiu}) + h(\text{Sibiu}) = 338 + 253 = 591$
- $f(\text{Bucharest}) = c(\text{Fagaras, Bucharest}) + h(\text{Bucharest}) = 450 + 0 = 450$
Best choice is Pitesti !!!

A* search example

Expand Pitesti and determine $f(n)$ for each node:
- $f(\text{Bucharest}) = c(\text{Pitesti, Bucharest}) + h(\text{Bucharest}) = 418 + 0 = 418$
Best choice is Bucharest !!!

Note values along optimal path !!
BUT ... graph search

Discards new paths to repeated state.
- Previous proof breaks down
Solution:
- Add extra bookkeeping i.e. remove more expensive of two paths.
- Ensure that optimal path to any repeated state is always first followed.
- Extra requirement on h(n): consistency (monotonicity)

Consistency

A heuristic is consistent if
\[ h(n) \leq c(n, a, n') + h(n') \]
If h is consistent, we have
\[ f(n') = g(n') + h(n') \]
\[ = g(n) + c(n, a, n') + h(n') \]
\[ \geq g(n) + h(n) \]
\[ \geq f(n) \]
i.e. f(n) is nondecreasing along any path.

Optimality of A* (more useful)

A* expands nodes in order of increasing f value
Contours can be drawn in state space
- Uniform-cost search adds circles.
- F-contours are gradually added:
  1) Nodes with f(n)<C*
  2) Some nodes on the goal
Contour f(n)=C*

Contour I has all nodes with f=fi, where fi < fi+1.

A* search, evaluation

Completeness: YES
- Since bands of increasing f are added
- Unless there are infinitely many nodes with f<f(G)

Time complexity:
- Number of nodes expanded is still exponential in the length of the solution.

Space complexity:
- It keeps all generated nodes in memory
- Hence space is the major problem not time
A* search, evaluation

Completeness: YES
Time complexity: (exponential with path length)
Space complexity: (all nodes are stored)
Optimality: YES
- Cannot expand $f_i$, until it 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 optimally efficient (not including ties)

Recursive best-first search

function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
return RFBS(problem, MAKE-NODE(INITIAL-STATE[problem]), $\infty$)
function RFBS(problem, node, f_limit) return a solution or failure and a new $f$-cost limit
if GOAL-TEST[problem](STATE[node]) then return node
successors ← EXPAND(node, problem)
if successors is empty then return failure, $\infty$
for each $s$ in successors do
$f(s) \leftarrow \text{max}(g(s) + h(s), f(node))$
repeat
best ← the lowest $f$-value node in successors
if $f(best) > f_limit$ then return failure, $f(best)$
alternative ← the second lowest $f$-value among successors
result, $f(best)$ ← RFBS(problem, best, min($f_limit$, alternative))
if result ≠ failure then return result

Memory-bounded heuristic search

Some solutions to A* space problems (maintain completeness and optimality)
- Iterative-deepening A* (IDA*)
  - Here cutoff information is the $f$-cost ($g+h$) instead of depth
- Recursive best-first search (RBFS)
  - Recursive algorithm that attempts to mimic standard best-first search with linear space.
  - (simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full

Recursive best-first search, ex.

Path until Rumnicu Vilcea is already expanded
Above node: $f$-limit for every recursive call is shown on top.
Below node: $f(n)$
The path is followed until Pitesti which has a $f$-value worse than the $f$-limit.

Recursive best-first search, ex.

Unwind recursion and store best $f$-value for current best leaf Pitesti
result, $f$(best) ← RBFS(problem, best, min($f$-limit, alternative))
best is now Fagaras. Call RFBS for new best
  - best value is now 450
Recursive best-first search, ex.

Unwind recursion and store best $f$-value for current best leaf.

Fagaras result: ($\text{best} = \text{RBFS}(\text{puzzle}, \text{best}, \text{min}(\text{f-limit}, \text{alternative}))$)

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

- Subtree is again expanded.
- Best alternative subtree is now through Timisoara.

Solution is found since $447 > 417$.

RBFS evaluation

RBFS is a bit more efficient than IDA*.

- Still excessive node generation (mind changes)
- Like A*, optimal if $h(n)$ is admissible
- Space complexity is $O(bd)$.

Time complexity difficult to characterize.

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

IDA* and RBFS suffer from too little memory.

(simplified) memory-bounded A*

Use all available memory.

- i.e. expand best leafs until available memory is full
- When full, SMA* drops worst leaf node (highest $f$-value)
- Like RBFS backup forgotten node to its parent

What if all leafs have the same $f$-value?

- Same node could be selected for expansion and deletion.
- SMA* solves this by expanding newest best leaf and deleting oldest worst leaf.

SMA* is complete if solution is reachable, optimal if optimal solution is reachable.

Learning to search better

All previous algorithms use fixed strategies.

Agents can learn to improve their search by exploiting the meta-level state space.

- Each meta-level state is a internal (computational) state of a program that is searching in the object-level state space.
- In A* such a state consists of the current search tree.

A meta-level learning algorithm from experiences at the meta-level.

Heuristic functions

E.g for the 8-puzzle

- Avg. solution cost is about 22 steps (branching factor +/- 3)
- Exhaustive search to depth 22: $3.1 \times 10^{10}$ states.
- A good heuristic function can reduce the search process.

Heuristic functions

E.g for the 8-puzzle knows two commonly used heuristics:

$h_1 = \text{the number of misplaced tiles}$

- $h_1(s) = 8$

$h_2 = \text{the sum of the distances of the tiles from their goal positions (manhattan distance)}$

- $h_2(s) = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
Heuristic quality

Effective branching factor $b^*$
- Is the branching factor that a uniform tree of depth $d$ would have in order to contain $N+1$ nodes.
  \[ N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \]
- Measure is fairly constant for sufficiently hard problems.
- Can thus provide a good guide to the heuristic’s overall usefulness.
- A good value of $b^*$ is 1.

Inventing admissible heuristics

Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:
- Relaxed 8-puzzle for $h_1$: a tile can move anywhere.
  As a result, $h_1(n)$ gives the shortest solution.
- Relaxed 8-puzzle for $h_2$: a tile can move to any adjacent square.
  As a result, $h_2(n)$ gives the shortest solution.

The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.

ABSolver found a useful heuristic for the Rubik’s cube.

Inventing admissible heuristics

Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
This cost is a lower bound on the cost of the real problem.
Pattern databases store the exact solution to every possible subproblem instance.
- The complete heuristic is constructed using the patterns in the DB.

Inventing admissible heuristics

Another way to find an admissible heuristic is through learning from experience:
- Experience = solving lots of 8-puzzles.
- An inductive learning algorithm can be used to predict costs for other states that arise during search.

Local search and optimization

Previously: systematic exploration of search space.
- Path to goal is solution to problem.
YET, for some problems path is irrelevant.
- E.g 8-queens.

Different algorithms can be used:
- Local search.
**Local search and optimization**

Local search = use single current state and move to neighboring states. 
Advantages:
- Use very little memory
- Find often 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-climbing search**

"Is a loop 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 a.k.a. greedy local search

**Hill-climbing example**

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 (directly or indirectly).

**Hill-climbing search**

```plaintext
function HILL-CLIMBING(problem) return a state that is a local maximum
input: problem, a problem
local variables: current, a node.
neighbor, a node.

current ← MAKE-NODE(INITIAL-STATE[problem])

loop do
neighbor ← a highest valued successor of current
if VALUE[neighbor] ≤ VALUE[current] then return
STATE[current]
current ← neighbor
```

**Hill-climbing example**

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 ($h=1$).
Drawbacks

Ridge = sequence of local maxima difficult for greedy algorithms to navigate Plateaux = an area of the state space where the evaluation function is flat. Gets stuck 86% of the time.

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
- cfr. 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.

Simulated annealing

Escape local maxima by allowing "bad" moves.
- Idea: but gradually decrease their size and frequency.

Origin; metallurgical annealing

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.

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 at proportionally to state success.

Genetic algorithms

Variant of local beam search with sexual recombination.
Genetic algorithms

Variant of local beam search with sexual recombination.

AI 1 Genetic algorithm

function GENETIC_ALGORITHM(population, FITNESS-FN) return an individual
input: population, a set of individuals
FITNESS-FN, a function which determines the quality of the individual
repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
        x ← RANDOM_SELECTION(population, FITNESS_FN)
        y ← RANDOM_SELECTION(population, FITNESS_FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then
            child ← MUTATE(child)
        add child to new_population
    population ← new_population
until some individual is fit enough or enough time has elapsed
return the best individual

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,…

Online search problems

Agent knowledge:
- ACTION(s): list of allowed actions in state s
- C(s,a,s'): step-cost function (1 After s' is determined)
- GOAL-TEST()
An agent can recognize previous states.
Actions are deterministic.
Access to admissible heuristic h(s)
e.g. manhattan distance

Objective: reach goal with minimal cost
- Cost = total cost of travelled path
- Competitive ratio=comparison of cost with cost of the solution path if search space is known.
- Can be infinite in case of the agent accidentally reaches dead ends

The adversary argument

Assume an adversary who can construct the state space while the agent explores it
- Visited states S and A. What next?
- Falls in one of the state spaces
No algorithm can avoid dead ends in all state spaces.
Online search agents

The agent maintains a map of the environment.
- Updated based on percept input.
- This map is used to decide next action.

Note difference with e.g. A*
An online version can only expand the node it is physically in (local order)

Online DF-search

```
function ONLINE_DFS-AGENT(s') return an action
input: s', a percept identifying current state
static: result, a table indexed by action and state, initially empty
static: unexplored, a table that lists for each visited state, the action not yet tried
static: unbacktracked, a table that lists for each visited state, the backtrack not yet tried
s, the previous state and action, initially null

if GOAL-TEST(s') then return stop
if s' is a new state then do
    unexplored[s'] ← ACTIONS(s')
    add s' to the front of unbacktracked
end
if unexplored[s'] is empty then
    if unbacktracked[s'] is empty then return stop
    else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
    else a ← POP(unexplored[s'])
    s ← s'
return a
```

Online DF-search, example

Assume maze problem on 3x3 grid.
 s' = (1,1) is initial state
Result, unexplored (UX), unbacktracked (UB), ... are empty
S,a are also empty

Online DF-search, example

GOAL-TEST((2,1))?
- s not = G thus false
- (2,1) a new state?
  - True
  - ACTION(2,1) -> UX[(2,1)]
    (continued)
- s is null?
  - False
  - result[UP(1,1)] < (2,1)
  - UB[(2,1)]={(1,1)}
  - UX[(2,1)] empty?
    - False
  - A=DOWN, s=(2,1) return A

Online DF-search, example

GOAL-TEST((1,1))?
- s not = G thus false
- (1,1) a new state?
  - True
  - ACTION(1,1) -> UX[(1,1)]
    (continued)
- s is null?
  - True (initially)
  - UX[(1,1)] empty?
    - False
  - POP(UX[(1,1)])→a
    - A=UP
  - s = (1,1)
  Return a
Online DF-search, example

GOAL-TEST((1,2))?
- S not = G thus false
- True, UX[(1,2)]=LEFT,UP

UX[(1,2)]=LEFT,UP
S'=(1,2)

s is null?
- false (s=(1,1))
- result[LEFT,(1,1)] <- (1,2)
- UB[(1,2)]={(1,1)}

UX[(1,2)] empty?
- False
- UB[1,1] empty? False
A=LEFT, s=(1,2) return A

S'=(1,2)

Online DF-search

Worst case each node is visited twice.
An agent can go on a long walk even when it is close to the solution.
An online iterative deepening approach solves this problem.
Online DF-search works only when actions are reversible.

S'=(1,1)

Online local search

Hill-climbing is already online
- One state is stored.
Bad performance due to local maxima
Solution: Random restarts impossible.

Learning real-time A*

function LRTA*-COST(s,a,s',H) return an cost estimate
if s' is undefined return Estimated(h(s))
else return Estimated(c(s,a,s') + H[s'])

function LRTA*-AGENT(s') return an action
input: s', a percept identifying current state
static: result, a table indexed by action and state, initially empty
H, a table of cost estimates indexed by state, initially empty
s,a, the previous state and action, initially null

if GOAL-TEST(s') then return stop
if s' is a new state (not in H) then H[s'] <- Estimated(h(s'))

unless s is null
result(s,a) = Estimated(c(s,a,s') + H[s'])
H[s'] <- Estimated(c(s,a,s') + H[s'])

a = an action s in ACTIONS(s') that minimizes LRTA*-COST(s',result(a,s'),H)
s = a
return a

Solution 2: Add memory to hill climber
- Store current best estimate H(s) of cost to reach goal
- H(s) is initially the heuristic estimate h(s)
- Afterward updated with experience (see below)

Learning real-time A* (LRTA*)