Lecture 7: Search 6

Victor R. Lesser
CMPSCI 683
Fall 2010

This Lecture

- RTA*
- Hierarchical A*
- Beginning of Local Search
  - Hill-Climbing/Iterative Improvement

RTA* - Real-Time A*

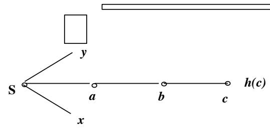
Intermix partial search with execution of action

- Goal: reduce the execution time of A*.
- Method: limit the search horizon of A* and select an action (single move) in constant time.
  - Make decision about *next move in real-world* without a *complete plan* (path) to reach goal state
- Two stages
  - Make individual move decision: Perform mini-min search with alpha pruning
  - Make a sequence of decisions to arrive at a solution
    - recovering from inappropriate actions
    - avoid loops

First Phase - Minimin Search with Alpha-Pruning

- Mini-min depth-first look-ahead search
  - Returns back-up f value for a node from looking ahead to the frontier node at the horizon
    - *Can viewed as simply a more accurate and computationally expensive heuristic function*
  - Reason: If the heuristic function *h* is consistent/monotone and admissible, **then the error in the backed-up cost estimate cannot increase with search depth, f is always increasing and thus better estimate of actual cost**
- Alpha pruning
  - If **current minimum f of horizon node (alpha value)** is less than f of an intermediate node, the intermediate node (and any successors) can be eliminated from further consideration
  - Reason: f is monotonic (*never can get lower f)* and you are only searching to horizon (don't need goal state to prune)

Basis of RTA*



- *$h(a) \leq g(a \text{ to } c) + h(c) \leq h^*(a)$;*
  - *assuming you need to go to the goal state thru c from a*

- As a result of exploring in the search space from *a* to *c*, you can replace *h*(*a*) with the *better (more informed) estimate g(a to c) + h(c)*

- This leads to a more informed decision at S whether to take the "action in the real world of moving" to either state *y*, *a*, or *x*.

Procedure for Calculating Backed-Up Value of a Move

procedure evaluate(move,*limit*)
/* return backed-up estimate *f´* (move) by ∝-pruning search to depth *limit* */

1. Open ← {move}; ∝← ∞
2. *f* (move) ← *g* (move) + *h* (move);
3. While (open not empty) **do**
4. node ← pop (Open);
5. expand node; for each child of node **do**
6. *g* (child) ← *g* (node) + *move-cost*;
7. *f* (child) ← *g* (child) + *h* (child);
   Prune child if *f* (child) >= ∝
8. if *f* (child) < ∝ **do**
9. if (depth = *limit* or goal(child)) then
   ∝← *f* (child);
10. else put child on Open; **od od od**
11. Return ∝;

RTA* - Controlling the Sequence of Moves Executed in Real-World

Basic Principle:

"One should backtrack to a *previously visited real world state* when the **estimate of solving the problem from that state plus the cost of returning to that state** is less than the estimated cost of going forward from the current state." - Korf

- Merit of every node f(n) = g(n) + h(n) is measured *relative to the current position* of the problem solver in the real-world
  - **initial state is irrelevant**

- If one moves back to a previously *visited real-world state*, then it needs to take into account that one already has taken action there
  - **value of state is next best *f***

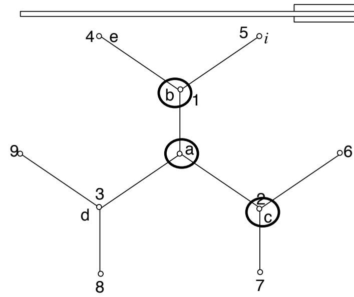
Remember Interplay between partial search and execution of action in real-world

RTA* Algorithm

- Maintains in a hash table a list of those states/nodes that have been visited by an *actual move* in the real world of the problem solver;
- At each cycle in the real-world, the current state is expanded and the heuristic function, possibly augmented by look-ahead search, is applied to each successor state which is **not in the hash table**;
- The *f* value of each neighboring state is computed by adding the *h* value plus the cost of the link to the current state;
- The neighbor with the minimum *f* value is chosen for the current state;
- The **second best *f* value is stored in the hash table** for the current state
  - Represents the estimated *h* cost of solving the problem by returning to this state
  - Second best avoids loops

Example of RTA*

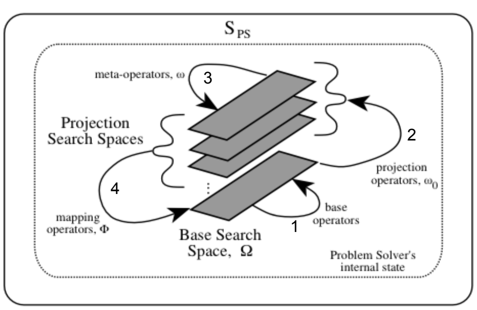
node	h	f
a	3	4
b	5	2
c	2	3
d	3	4
e	4	5
i	5	6



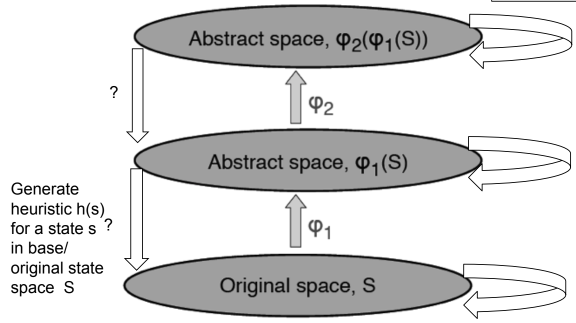
Characteristics of RTA*

- Completeness of RTA*
  - In a finite problem space with positive edge costs and finite heuristic value, in which a goal state is reachable from every state, RTA* will find a solution.
- Local optimal of RTA*
  - Each move made by RTA* on a tree is along a path whose estimated cost of reaching a goal is minimum, based on the cumulative search frontier at the time.

Hierarchical Problem Solving



Hierarchical Heuristic A* Search

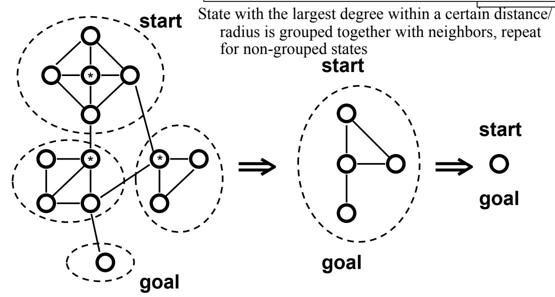


Automatically Generating State Space Abstraction "Max-degree" Star Abstraction

- The state with the highest degree is grouped together with its neighbors within a certain distance (the abstraction radius) to form a single abstract state.

Star abstraction with radius = 1

State with the largest degree within a certain distance/ radius is grouped together with neighbors, repeat for non-grouped states



Naive Hierarchical A*

TABLE 1. Naive Hierarchical A*. (abstraction radius = 2)

Search Space	Size (# states) All Levels	Size (# states) Base Level	Nodes Expanded: Blind Search	Hierarchical A* All Levels	Hierarchical A* Base Level
Blocks-5	1166	866	389	2766	118
5-puzzle	961	720	348	3119	224
Fool's Disk	4709	4096	1635	12680	629
Hanoi-7	2894	2187	1069	18829	701
KL2000	3107	2736	1236	7059	641
MC 60-40-7	2023	1878	934	2412	702
Permute-6	731	720	286	806	77
Words	5330	4493	1923	19386	604

Naïve Hierarchical A* - Cache h in abstract space; avoid search for h(s2) if h(s1) already computed [h(Φ(s1))] and *Φ(s1) = Φ(s2)*

Reducing Search in Abstract Spaces

- Observation: all searches related to the same base level problem have the same goal.
- This allows additional types of caching of values.
- It leads to variants of Hierarchical A* Search (Valtorta's barrier) requiring less effort in 5 out of 8 search spaces.

Exploit Information for Repeated Blind Search in Abstract Space

- V1 - h*caching
  - Cache exact h's (h*) along optimal solution in abstract space
    - Cache for use in base level search (don't need to search again since already know optimal distant to goal in abstract space) [$h(\Phi(s1))$, $h(\Phi_i)$.. [$h(\Phi_j)$)]
- V2
  - Cache optimal path in abstract space (optimal-path caching)
    - Exploit in further searches in abstract space (if reach such a node in abstract space can stop search along this path) – can stop further search in Φ once you have $h(\Phi_i)$ that you found was on optimal path in previous search
- V3
  - pertains to states that were opened (or closed) during abstract search but are not on the solution path
  - Remember optimal path length in abstract search space (P-g caching)
    - P being optimal path length from start to goal in abstract space

Hierarchical A*

TABLE 2. Hierarchical A*. (abstraction radius = 2)

Search Space	Nodes Expanded: Blind Search	Hierarchical A*: Naive	V1	V2	V3	# problems V3 < BS (out of 200)
Blocks-5	389	2766	1235	478	402	96 *
5-puzzle	348	3119	1616	854	560	14 *
Fool's Disk	1635	12680	8612	3950	**1525**	132
Hanoi-7	1069	18829	10667	5357	3174	0 *
KL2000	1236	7059	3490	1596	**1028**	171
MC 60-40-7	934	2412	1531	1154	**863**	128
Permute-6	286	806	482	**279**	**242**	113
Words	1923	19386	7591	2849	**1410**	124

The Granularity of Abstraction

- Increasing the radius of abstraction has two contradictory effects:

  + abstract spaces contain fewer states and each abstract search produces values for more states, but

  - the heuristic is less discriminating
- Using the best case radius Hierarchical A* Search (Valtorta's barrier) is more effective every search space.

Hierarchical A* with best abstraction radius

TABLE 3. Hierarchical A*. (best abstraction radius)

Search Space	Radius	Nodes Expanded: Blind Search	Hierarchical A*: Naive	Hierarchical A*: V3	# problems V3 < BS (out of 200)	CPU seconds: Blind Search	CPU seconds: V3
Blocks-5	5	389	611	**309**	123	69	86
5-puzzle	12	348	354	**340**	131	36	40
Fool's Disk	4	1635	**1318**	**1172**	194	872	902
Hanoi-7	20	1069	1097	**1055**	117	102	108
KL2000	5	1236	1306	**1072**	178	398	384
MC 60-40-7	4	934	**822**	**803**	144	266	253
Permute-6	5	286	**201**	**194**	192	82	67
Words	3	1923	9184	**1356**	128	1169	1273

Local Search

- In many optimization problems, path is irrelevant (no path cost); the goal state itself is the solution
  - 8-queens problem, job-shop scheduling
  - circuit design, computer configuration
  - automatic programming, automatic graph drawing
- Then state space = set of “complete” configurations; need to find optimal configuration
- Can use iterative improvement algorithms; keep single “current” state and try to improve it
  - *Paths followed by search are not retained*
    - Contrast with open and closed node lists; search tree

Advantages of local search

- Very simple to implement.
- Very little memory is needed.
- Can often find reasonable solutions in very large (*continuous*) state spaces for which systematic algorithms are not suitable.

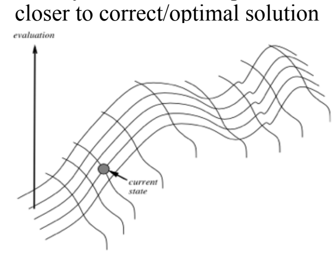
Stochastic vs. Systematic Search

- Unsolvability -- Is there a solution?
  - Systematic: can require exhaustive examination of exponential search space
  - Stochastic: cannot determine unsolvability
- Completeness/Optimality
  - Systematic: complete
  - Stochastic: incomplete
- Speed
  - Neither is uniformly superior; each does better for different sorts of problems

Local Search is an example of Stochastic Search

Iterative Improvement
(Smart version of Generate & Test)

- Start Search with complete but non-optimal solution
- Modify incorrect/non-optimal solution to move it closer to correct/optimal solution



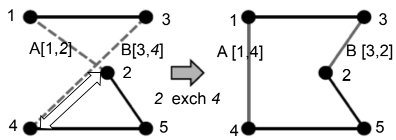
Path Cost Minimization

versus

Value optimization

Example: Traveling Salesperson Problem

- Start with any complete tour and perform pair wise exchanges of the end points of two segments



- Only make change if exchange reduces tour cost
- Variants of this approach get within 1% of optimal very quickly with thousands of cities

Next Lecture

- Continuation of Local Search
  - Hill-Climbing/Iterative Improvement
    - Simulated Annealing (Stochastic Hill Climbing)
  - Beam Search
    - Genetic Algorithm
  - Repair/Debugging (to be done next time)
    - GSAT