

Lecture 6: Search 5

Victor R. Lesser
CMPSCI 683
Fall 2010

This lecture

- ◆ Other Time and Space Variations of A*
 - Finish off RBFS
 - SMA*
 - Anytime A*
 - RTA* (maybe if have time)

V. Lesser: CS683, F08

RBFS - Recursive Best-First Search

- ◆ Mimics best-first search with linear space
- ◆ Similar to recursive depth-first
 - Limits recursion by keeping track of the f -value of the best alternative path from any ancestor node – *one step look-ahead*
 - *If current node exceeds this value, recursion unwinds back to the alternative path – same idea as contour*
 - As recursion unwinds, replaces f -value of node with *best f -value of children*
 - *Allows to remember whether to re-expand path at later time*
- ◆ Exploits information gathered from previous searches about minimum f so as to focus further searches

V. Lesser: CS683, F08

RBFS - Recursive Best-First Search Algorithm

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) returns a solution, or failure
RBFS(MAKE-NODE(INITIAL-STATE[*problem*]), ∞)

function RBFS(*problem*, *node*, *f-limit*) returns a solution, or failure and a new f -cost limit

if GOAL-TEST(*problem*)(*state*) then return *node*

successors ← EXPAND(*node*, *problem*)

if successors is empty, then return failure, ∞

for each s in successors do $f[s] ← \max(g(s) + h(s), f[node])$: Pathmax heuristic; guarantee monotonic f

repeat

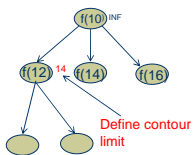
best ← the lowest f -value node in successors

if $f[best] > f-limit$ then return failure, $f[best]$

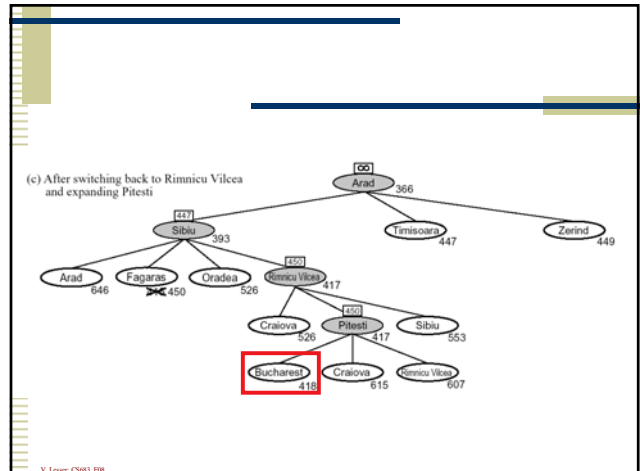
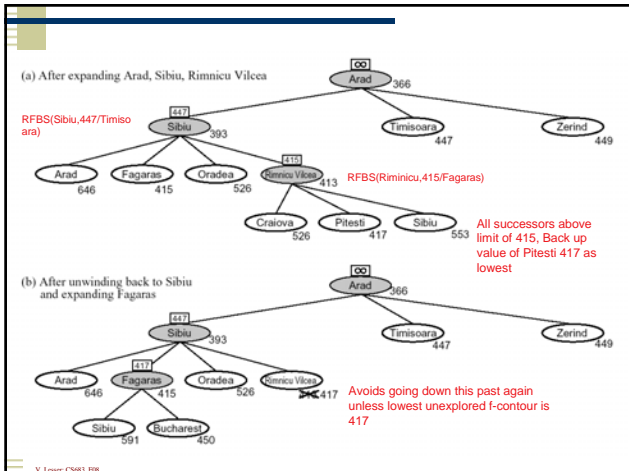
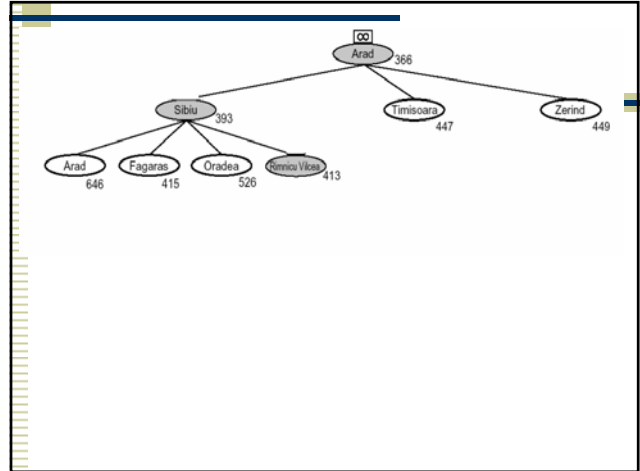
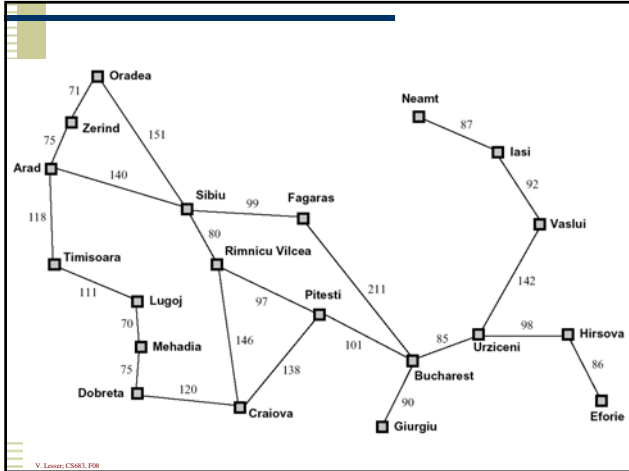
alternative ← the second-lowest f -value among successors

result, $f[best] ←$ RBFS(*problem*, *best*, $\min(f-limit, alternative)$) Recursive search on best successor, remember when to backup

end



V. Lesser: CS683, F08



RBFS -- Pro's and Con's

- ◆ More efficient than IDA* and still optimal
 - Best-first Search based on **next best f-contour**; fewer regeneration of nodes
 - Exploit results of search at a specific f-contour by saving next f-contour associated with a node whose successors have been explored.
- ◆ Like IDA* still suffers from excessive node regeneration
- ◆ IDA* and RBFS not good for graphs
 - Can't check for repeated states other than those on current path
- ◆ Both are hard to characterize in terms of expected time complexity

V. Lesser, CS683, F08

SMA*(Simplified Memory-Bounded A*)

- ◆ Uses a given amount of memory to remember nodes so that they don't have to be repeatedly regenerated
- ◆ It will utilize whatever memory is made available to it.
- ◆ It avoids repeated states as far as its memory allows.

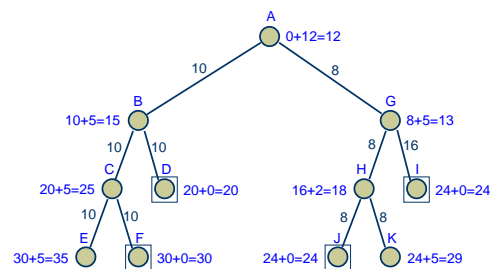
V. Lesser, CS683, F08

SMA*

- Expand **deepest lowest f-cost leaf-node**
 - *Best first search on f-cost*
- **Update** f-cost of nodes whose **successors have higher f-cost**
- Drop shallowest & **highest** f-cost leaf node
 - **remember best forgotten descendant**
- Paths longer than node limit get ∞ cost.

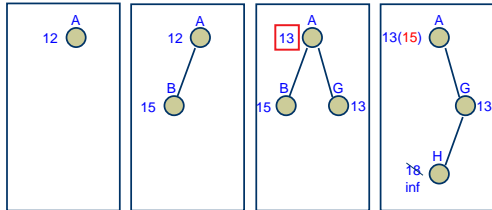
V. Lesser, CS683, F08

SMA* Example



V. Lesser, CS683, F08

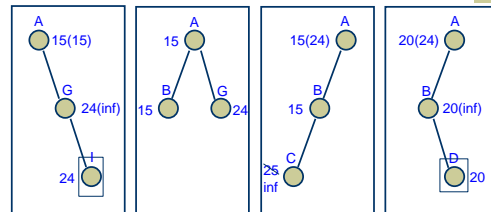
SMA* Example (3-node limit)



Update A based on lowest cost f successor?
Remember next lowest cost f node B that is removed

V. Lesser, CS883, F08

SMA* Example (3-node limit) cont.



Reach goal node I but it is not the cheapest so continue search
Regenerate node B, remember that there was node with $f=15$ that had been removed, remember successor of G has $f=24$
C is not goal node and it is at max depth
Why don't we need to search anymore after finding D.

V. Lesser, CS883, F08

SMA* Analysis

- It is complete, provided the available memory is sufficient to store the shallowest solution path.
- It is optimal, if enough memory is available to store the shallowest optimal solution path. Otherwise, it returns the best solution (if any) that can be reached with the available memory.
- Can keep switching back and forth between a set of candidate solution paths, only a few of which can fit in memory (**thrashing**)

Memory limitations can make a problem intractable wrt time

- With enough memory for the entire tree, same as A*

Sketch of SMA* Algorithm

```

function SMA*(problem) returns a solution sequence
inputs: problem, a problem
static: Queue, a queue of nodes ordered by f-cost
Queue ← MAKE-QUEUE((MAKE-NODE(INITIAL-STATE[problem])))
loop do
  if Queue is empty then return failure
  n ← deepest least-f-cost node in Queue ; same as A* except deepest criterion added
  if GOAL-TEST(n) then return success ; generate one successor at time
  s ← NEXT-SUCCESSOR(n)
  if s is not a goal and is at maximum depth then ; discard node which can't reach the
    f(s) ← ∞ ; goal in size of memory
  else
    f(s) ← MAX(f(n), g(s)+h(s)) ; remember what you have done
    if all of s's successors have been generated then ; by posting resulting higher in tree
      update s's f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
  if memory is full then
    delete shallowest, highest-f-cost node in Queue
    remove it from its parent's successor list
    insert its parent on Queue if necessary ; parent could have been removed previously
    insert s on Queue
end
    
```

V. Lesser, CS883, F08

Memory-bounded heuristic search

- ♦ IDA* - Iterative-deepening A*
 - Use f-cost ($g+h$) as cutoff
 - At each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration
- ♦ Recursive best-first search (RBFS)
 - Best-first search with only linear space
 - Keep track of the f-value of the best alternative
 - As the recursion unwinds, it forgets the sub-tree and back-up the f-value of the best leaf as its parent's f-value.
- ♦ SMA* proceeds like A*
 - Expanding the best leaf until memory is full
 - Drop the worst leaf node, and back-up the value of the forgotten node to its parent.
 - Complete IF there is any reachable solution.
 - Optimal IF any optimal solution is reachable.

V. Lessor, CS683, F08

Approaches for Reducing Search Cost

- ♦ **Staged search** involves periodically pruning unpromising paths
 - SMA* is an example of a staged search
- ♦ Node expansion may be so costly (because the branching factor is high or the cost to apply operators is high) that *exhaustive node expansion* is not practical.

V. Lessor, CS683, F08

Heuristic node expansion

- ♦ Use a generator approach to incrementally produce successors ordered by quality (must have *operator-ordering function*);
- ♦ Limit expansion so that only *likely* successors are generated (often called *plausible-move generator*);
- ♦ Prune unpromising successors immediately following node expansion;
- ♦ Delay state computation until expansion time when possible (must be able to compute h without state only on operator/previous state)

V. Lessor, CS683, F08

Real-Time Concerns

Real-time problem solving

- ◆ Practical and theoretical difficulties:
 - Agents have limited computational power.
 - They must react within an acceptable time.
 - Computation time normally reduces the value of the result.
 - There is a high degree of uncertainty regarding the rate of progress.
 - The “appropriate” level of deliberation is situation dependent.

V. Lesser: CS583, F08

Simon's “Bounded-Rationality”

“A theory of rationality that does not give an account of problem solving in the face of complexity is sadly incomplete. It is worse than incomplete; it can be seriously misleading by providing “solutions” that are without operational significance”

“The global optimization problem is to find the least-cost or best-return decision, net of computational costs.”

-- Herbert Simon, 1958

V. Lesser: CS583, F08

Satisficing

- ◆ A Scottish word which means satisfying.
- ◆ Denotes decision making that searches until an alternative is found that is satisfactory by the agent's aspiration level criterion.
- ◆ Heuristic search as satisficing.
- ◆ Formalizing the notion of satisficing.

V. Lesser: CS583, F08

Satisficing versus Optimizing

“It appears probable that, however adaptive the behavior of organisms in learning and choice situations, this adaptiveness falls far short of the ideal “maximizing” postulated in economic theory. Evidently, organisms adapt well enough to ‘satisfice’; they do not, in general, ‘optimize.’”

V. Lesser: CS583, F08

Optimizing in the Real-World

“In complex real-world situations, optimization becomes approximate optimization since the description of the real-world is radically simplified until reduced to a degree of complication that the decision maker can handle. Satisficing seeks simplification in a somewhat different direction, retaining more of the detail of the real-world situation, but settling for a satisfactory, rather than approximate-best, decision.”

- Which approach is preferable?

V. Lesser, CS683, F08

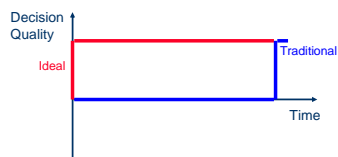
Anytime algorithms



- **Ideal** (maximal quality in no time)

V. Lesser, CS683, F08

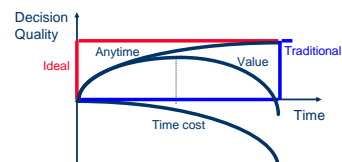
Anytime algorithms



- **Ideal** (maximal quality in no time)
- **Traditional** (quality maximizing)

V. Lesser, CS683, F08

Anytime algorithms



- **Ideal** (maximal quality in no time)
- **Traditional** (quality maximizing)
- **Anytime** (utility maximizing)

Value is a combination of quality of solution and amount of time to arrive at solution

V. Lesser, CS683, F08

Anytime A*

- A* is best first search with $f(n) = g(n) + h(n)$
- Three changes make it an anytime algorithm:
 - (1) Use a non-admissible heuristic so that sub-optimal solutions are found quickly.
 - (2) Continue the search after the first solution is found using it to prune the open list ?????
 - (3) When the open list is empty, the best solution generated is optimal.
- How to choose a non-admissible heuristic?

V. Lesser: CS883, F08

Weighted evaluation functions

- Use $f^w(n) = (1 - w)g(n) + wh(n)$
- Higher weight on $h(n)$ tends to search deeper.
- Admissible if $h(n)$ is admissible and $w \leq 0.5$
 - Same relative node ordering as admissible heuristic
 $h(n) \geq wh(n) / (1 - w)$ and $f(n) = f^w(n) / (1 - w)$
- Otherwise, the search is non-admissible, but it normally finds solutions much faster.
- **An appropriate w makes possible a tradeoff between the solution quality and the computation time.**

V. Lesser: CS883, F08

Algorithm 1: Anytime-WA*

```

Input: A start node start
Output: Best solution found and error bound
begin
  g(start) ← 0, f(start) ← h(start), f'(start) ← w × h(start) keep track of real and non-admissible f
  OPEN ← {start}, CLOSED ← ∅, incumbent ← nil Incumbent represent best complete solution so far found
  while OPEN ≠ ∅ and not interrupted do
    n ← arg minx ∈ OPEN f'(x)
    OPEN ← OPEN \ {n}
    if incumbent = nil or f(n) < f(incumbent) then ; can prune node if it cannot be better than existing solution
      CLOSED ← CLOSED ∪ {n}
      foreach ni ∈ Successors(n) such that g(n) + c(n, ni) + h(ni) < f(incumbent) do
        if ni is a goal node then
          f(ni) ← g(ni) + c(n, ni), incumbent ← ni
        else if ni ∈ OPEN ∪ CLOSED and g(ni) > g(n) + c(n, ni) then
          g(ni) ← g(n) + c(n, ni), f(ni) ← g(ni) + h(ni), f'(ni) ← g(ni) + w × h(ni)
          if ni ∈ CLOSED then
            OPEN ← OPEN ∪ {ni} ; if have found shorter path to node update node and put it back in play
            CLOSED ← CLOSED \ {ni}
          else
            g(ni) ← g(n) + c(n, ni), f(ni) ← g(ni) + h(ni), f'(ni) ← g(ni) + w × h(ni)
            OPEN ← OPEN ∪ {ni}
  if OPEN = ∅ then error ← 0
  else error ← f(incumbent) - minx ∈ OPEN f(x) ; bound error
  output incumbent solution and error bound
end
    
```

Pseudocode of Anytime WA*

V. Lesser: CS883, F08

Could you prune open list after each new incumbent??

Pruning States in Anytime A*

- For each node, store real $f(n) = g(n) + h(n)$
 - $f(n)$ is the lower bound on the cost of the best solution path through n
- When find solution/goal node n_1
 - $f(n_1)$ is an upper bound of the cost of the optimal solution
 - Prune all nodes n on the open list that have **real $f(n) \geq f(n_1)$??**

V. Lesser: CS883, F08

Adjusting W Dynamically*

Suppose you had the following situations, how would you adjust w .

- ♦ the open list has gotten so large that you are running out of memory?
- ♦ you are running out of time and you have not yet reached an answer?
- ♦ there are a number of nodes on the open list whose h value is very small?