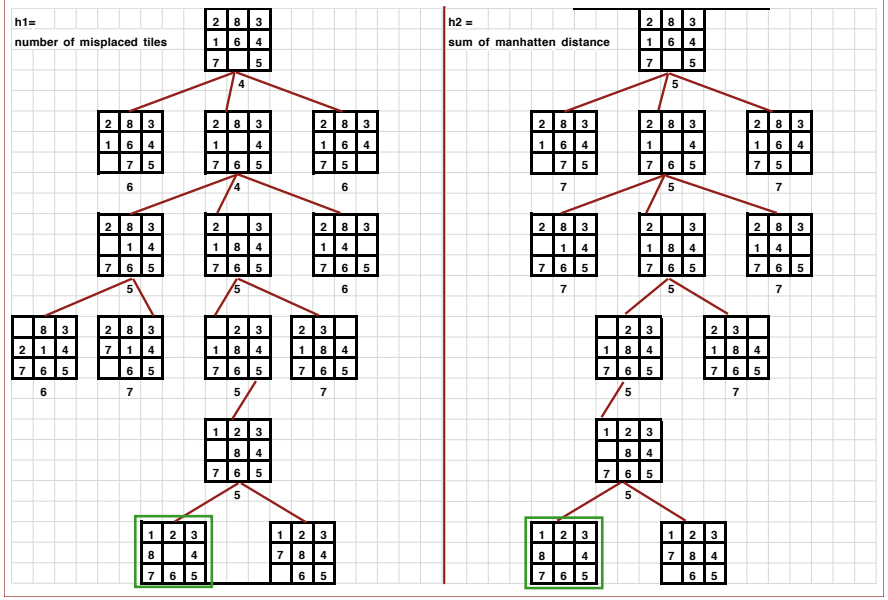
Lecture 4: Search - 3

Jiaying Shen

CMPSCI 683
Fall 2002

Example: tracing A* with two different heuristics



Today's lecture

- Space and time variations of A*

Search with limited memory

Problem: **How to handle the exponential growth of memory used by admissible search algorithms such as A*.**

Solutions:

- **IDA* [Korf, 1985]**
- **RBFS [Korf, 1993]**
- **SMA* [Russell, 1992]**

IDA* - Iterative deepening A*

- Beginning with an f-bound equal to the f-value of the initial state, perform a depth-first search bounded by the f-bound instead of a depth bound.
- Unless the goal is found, increase the f-bound to the lowest f-value found in the previous search that exceeds the previous f-bound, and restart the depth first search.

Iterative-Deepening-A*

- Algorithm: Iterative-Deepening-A*

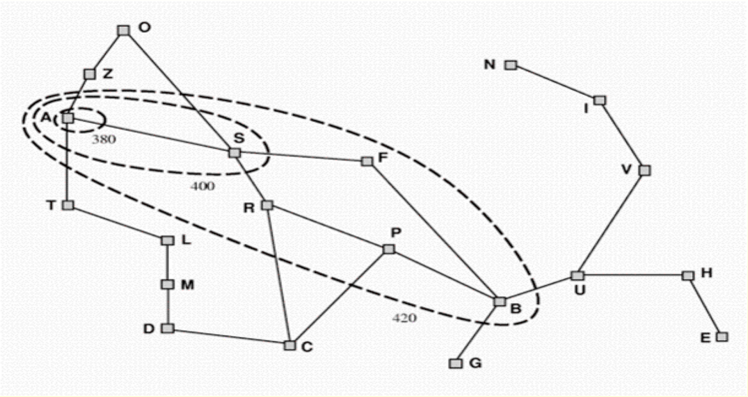
  1) Set THRESHOLD = the heuristic evaluation of the start state.

  2) Conduct a depth-first search based on minimal cost from current node, pruning any branch when its total cost function (g + h´) exceeds THRESHOLD. If a solution path is found during the search, return it.

  3) Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.

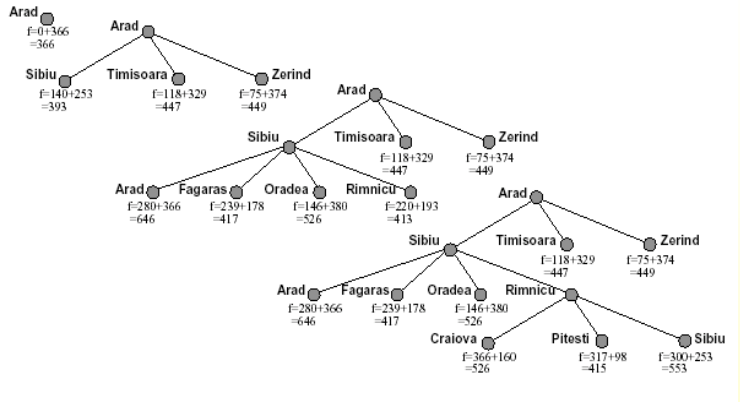
  - Start state always on path, so initial estimate is always overestimate and never decreasing.

f-Cost Contours

- 

- The more informed a heuristic, the more the contours will be "stretched" toward the goal (they will be more focused around the optimal path).

Stages in an IDA* Search for Bucharest



Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest...

What is the next Contour??

Experimental Results on IDA*

- IDA* is asymptotically same time as A* but only O(d) in space - versus $O(b^d)$ for A*
  - **Avoids overhead of sorted queue of nodes**
- IDA* is simpler to implement - no closed lists (limited open list).
- In Korf's 15-puzzle experiments IDA*: solved all problems, ran faster even though it generated more nodes than A*.
  - **A*: solved no problems due to insufficient space; ran slower than IDA***

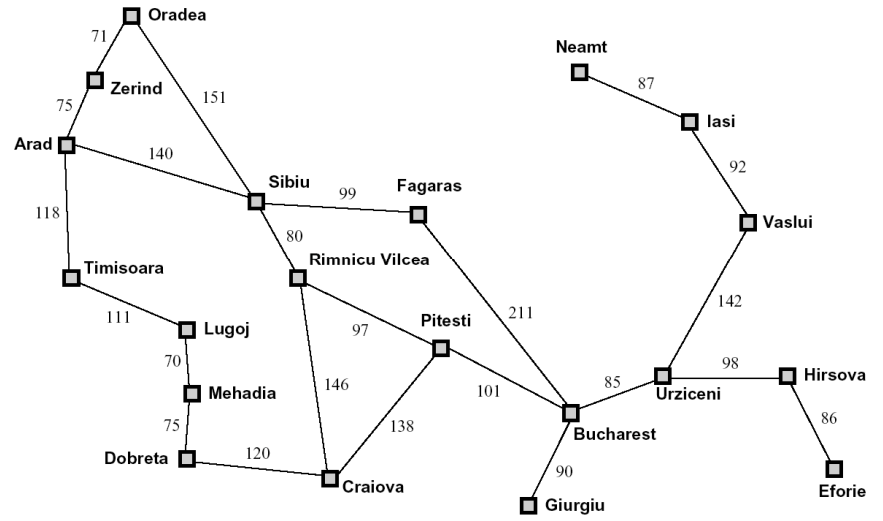
RBFS - Recursive Best-First Search

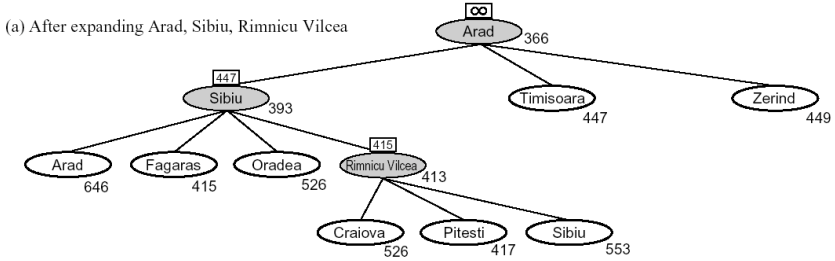
- **Mimics best-first search with linear space**
- **Similar to recursive depth-first**
  - **Limits recursion by keeping track of the f-value of the best alternative path from any ancestor node**
  - **If current node exceeds this value, recursion unwinds back to the alternative path**
  - **As recursion unwinds, replaces f-value of node with *best f-value* of children**
    - **Allows to remember whether to re-expand path at later time**

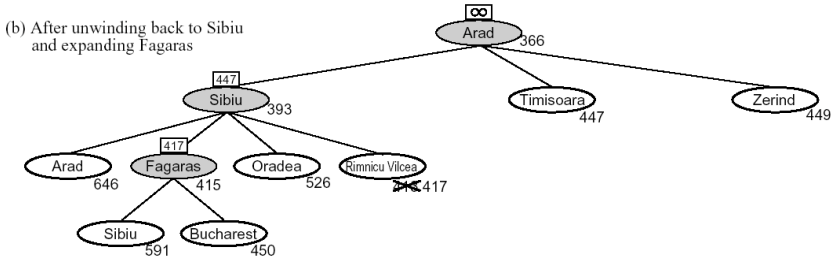
RBFS - Recursive Best-First Search Algorithm

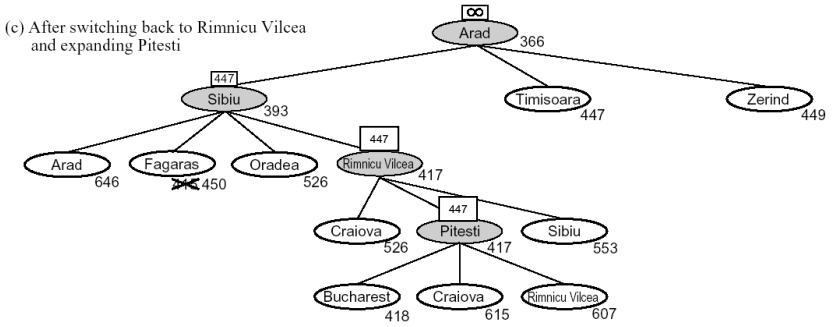
function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
RBFS(MAKE-NODE(INITIAL-STATE[*problem*]), ∞)

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 $\leftarrow$ EXPAND(*node*, *problem*)
if *successors* is empty, **then return** *failure*, ∞
for each s **in** *successors* **do** $f[s] \leftarrow \max(g(s) + h(s), f[node])$
repeat
best $\leftarrow$ the lowest f-value node in *successors*
if $f[best] > f\text{-}limit$ **then return** *failure*, $f[best]$
alternative $\leftarrow$ the second-lowest f-value among *successors*
result, $f[best] \leftarrow$ RBFS(*problem*, *best*, $\min(f\text{-}limit, alternative)$)
if *result* $\neq$ *failure* **then return** *result*
end









RBFS -- Pro's and Con's

- **More efficient than IDA* and still optimal**
- **Like IDA* suffers from excessive node regeneration**
  - Note mind changes in example
- **IDA* and RBFS not good for graphs**
  - Can't check for repeated states other than those on current path

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.**
- **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.**
- **When enough memory is available for the entire search tree, its behavior replicates that of A***

SMA*

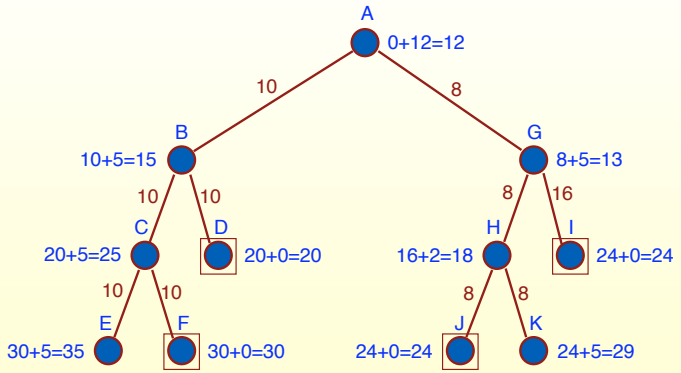
1. Expand deepest lowest *f*-cost leaf-node
2. Update *f* cost of nodes whose successors have higher *f*-cost
3. Drop shallowest & highest *f*-cost leaf node; remember best forgotten descendant
4. Paths longer than node limit get ∞ cost.

Sketch of SMA* Algorithm

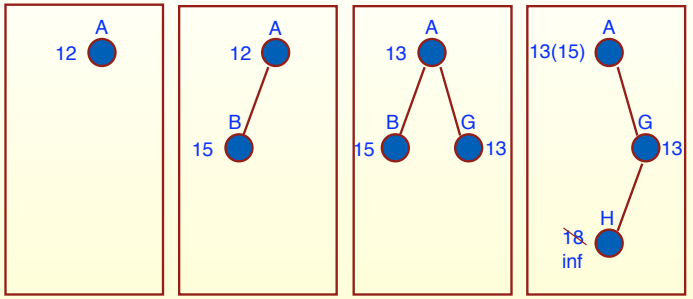
```
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  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
     if GOAL-TEST(n) then return success
     s ← NEXT-SUCCESSOR(n)
     if s is not a goal and is at maximum depth then
        f(s) ← ∞
     else
        f(s) ← MAX(f(n), g(s)+h(s))
     if all of n's successors have been generated then
        update n'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
     insert s on Queue
  end
```

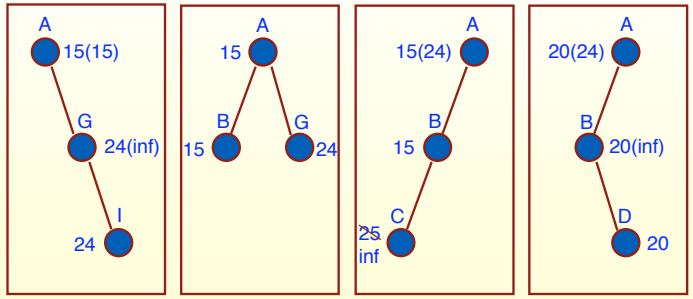

SMA* Example



SMA* Example (3-node limit)



SMA* Example (3-node limit) cont.



Why don't we need to search anymore after finding D.

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.**

Heuristic node expansion

- **Use a generator approach to incrementally produce successors in order 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)**

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.**

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

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.

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.' "

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?

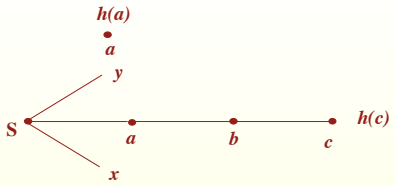
RTA* - Real-Time A*

- **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
  - Intermix partial search with execution of action
- **Two stages**
  - Make individual move decision: Perform minimin 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

- **Minimin lookhead search (see next slide)**
  - Returns back-up f value for a node from looking ahead to the frontier node at the horizon
  - Viewed as simply a more accurate and computationally expensive heuristic function
  - Reason: If the heuristic function *h* is consistent and admissible, then the error in the backed-up cost estimate cannot increase with search depth
- **Alpha pruning**
  - If current minimum f of horizon node is less than f of an intermediate node, the intermediate node (and any successors) can be eliminated from further consideration
  - Reason: f is monotonic and you are only searching to horizon (don't need goal state to prune)

Basis of RTA*



- ***h(a) ≤ g(a to c) + h(c) ≤ 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

Basic Principle:

"One should backtrack to a previously visited 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

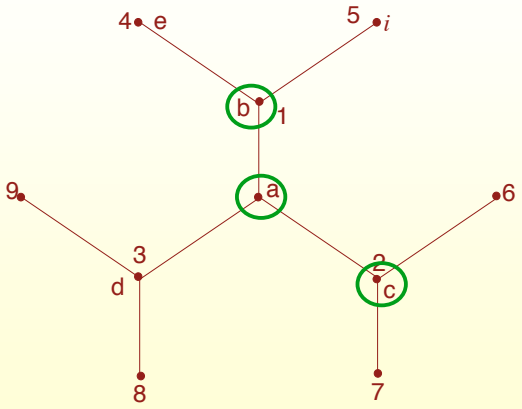
- **Merit of every node f(n) = g(n) + h(n) is measured relative to the current position of the problem solver in the real-world**
  - initial state is irrelevant
- **If one moves back to a previously visited real-world state, then it needs to take into account that one already has taken action there**
  - value of state is next best *f*

RTA* Algorithm

- **Maintains in a hash table a list of those nodes that have been visited by an actual move of the problem solver;**
- **At each cycle, the current state is expanded and the heuristic function, possibly augmented by look-ahead search, is applied to each state which is not in the hash table;**
- **The f value for 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	6
c	2	3
d	3	4
e	4	5
i	5	6



Next lecture

- **Anytime A***
- **Hierarchical A***