The Rich/Knight Implementation

- a node consists of
  - state
  - g, h, f values
  - list of successors
  - pointer to parent
- OPEN is the list of nodes that have been generated and had h applied, but not expanded and can be implemented as a priority queue.
- CLOSED is the list of nodes that have already been expanded.

Rich/Knight

```
1)   /* Initialization */

     OPEN <- start node

     Initialize  the start node
        g:
        h:
        f:

     CLOSED <- empty list
```

Rich/Knight

2) repeat until goal (or time limit or space limit)

- if OPEN is empty, fail
- BESTNODE <- node on OPEN with lowest f
- if BESTNODE is a goal, exit and succeed
- remove BESTNODE from OPEN and add it to CLOSED
- generate successors of BESTNODE

Rich/Knight

```
for each successor s do
   1. set its parent field
   2. compute g(s)
   3. if there is a node OLD on OPEN with
   the same state info as s
        { add OLD to successors(BESTNODE)
          if g(s) < g(OLD), update OLD and
             throw out s }
```

Rich/Knight/Tanimoto

```
4. if (s is not on OPEN and there is a node
     OLD on CLOSED with the same state
     info as s
     { add OLD to successors(BESTNODE)
         if g(s) < g(OLD), update OLD,
            remove it from CLOSED
           and put it on OPEN, throw out s

     }
```

Rich/Knight

```
5. If s was not on OPEN or CLOSED
   { add s to OPEN
     add s to successors(BESTNODE)
     calculate g(s), h(s), f(s) }



end of repeat loop
```

A* Extra Examples

- To show what happens when

1. It encounters a node whose state is already on OPEN
2. It encounters a node whose state is already on CLOSED

Thought Question

- Do you have to keep the list of successors for each node through the whole search?
- Rich/Knight did (why?)
- Tanimoto did not
- If you keep it, what might it be used for?

A* Example

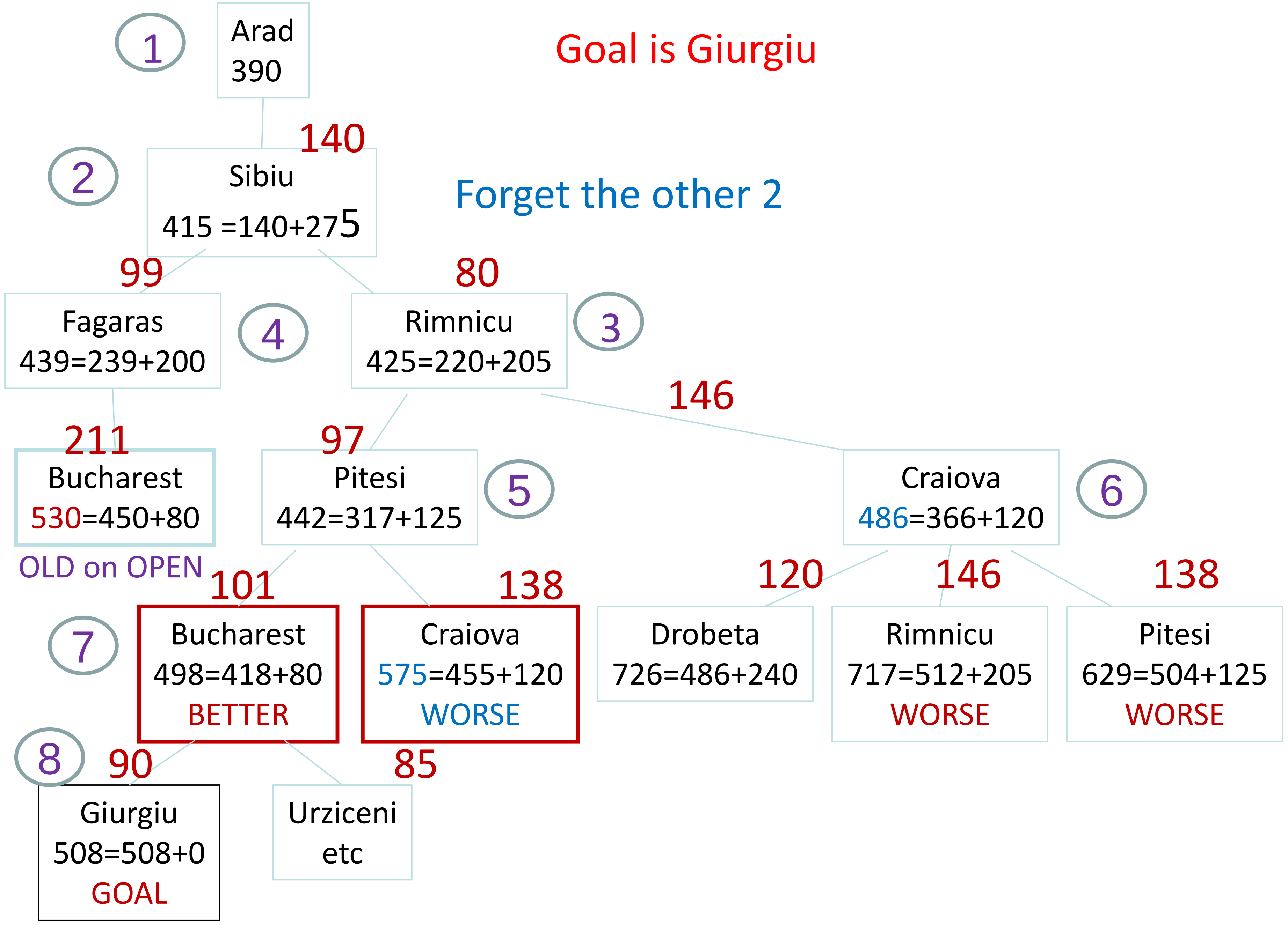
- Newly generated node s, but OLD on OPEN has the same state.
- Shortest path in Romania, but the goal is now Giurgiu, not Bucharest.

Straight line distances to Giurgiu (I made them up)

Arad	390
Sibiu	275
Fagaras	200
Rimnicu	205
Pitesi	125
Craiova	120
Bucharest	80
Drobeta	240

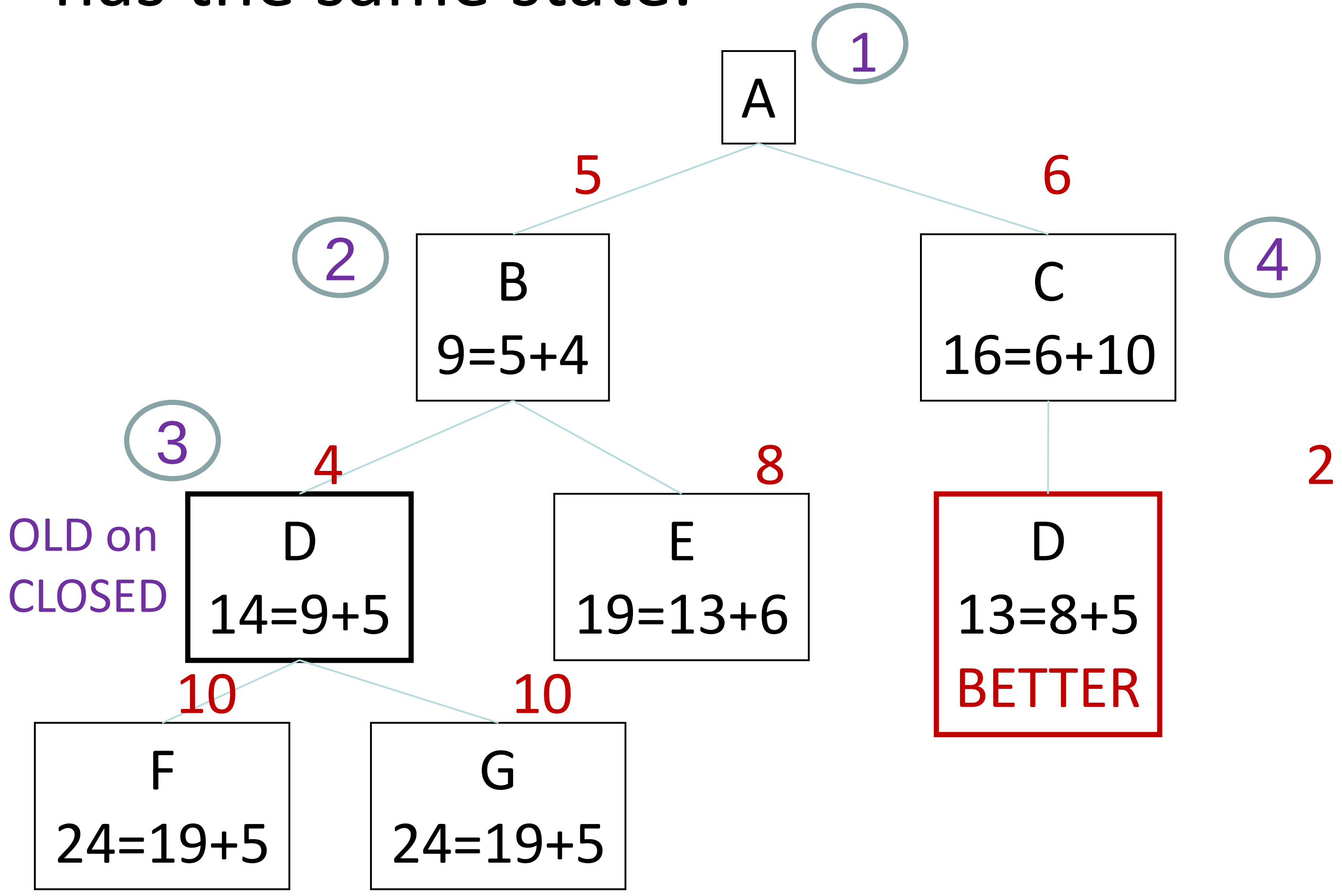
Goal is Giurgiu

Forget the other 2



A* Example (abstract, pretend it's time)

- Newly generated node s, but OLD on CLOSED has the same state.



The Heuristic Function **h**

- If h is a perfect estimator of the true cost then A* will always pick the correct successor with no search.
- If h is admissible, A* with TREE-SEARCH is guaranteed to give the optimal solution.
- If h is consistent, too, then GRAPH-SEARCH is optimal.
- If h is not admissable, no guarantees, but it can work well if h is not often greater than the true cost.

Complexity of A*

- Time complexity is exponential in the length of the solution path unless for "true" distance h*

  $|h(n) - h^*(n)| < \Theta(\log h^*(n))$

  which we can't guarantee.

- But, this is AI, computers are fast, and a good heuristic helps a lot.

- Space complexity is also exponential, because it keeps all generated nodes in memory.

Big Theta notation says 2 functions have about the same growth rate.

Why not always use A*?

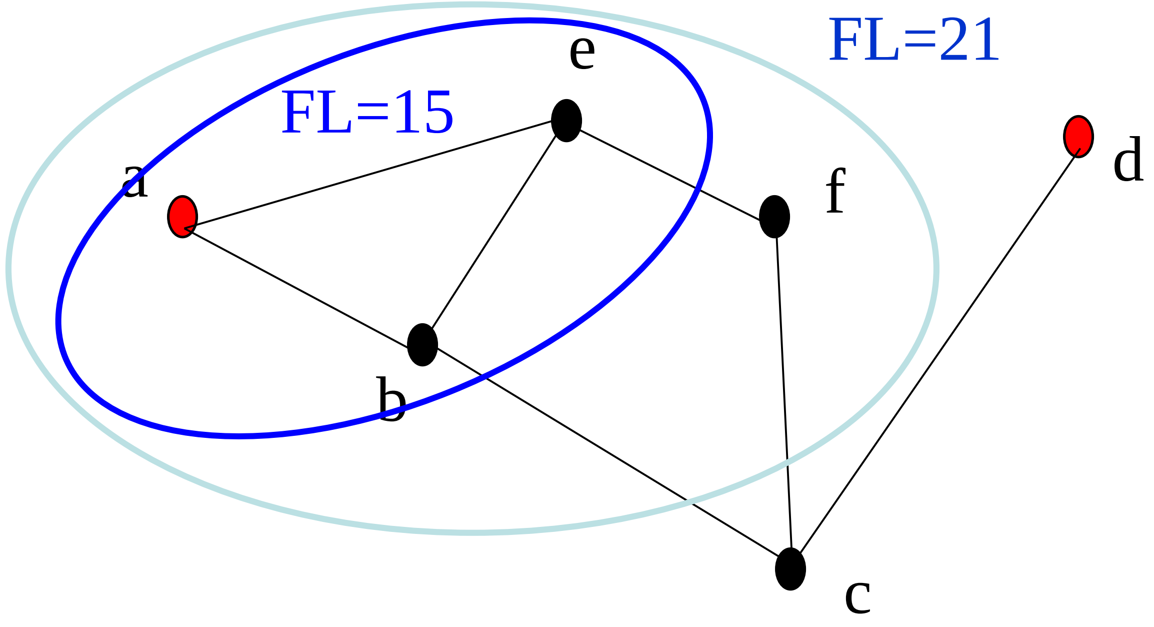
- Pros
- Cons

Solving the Memory Problem

- Iterative Deepening A*
- Recursive Best-First Search
- Depth-First Branch-and-Bound
- Simplified Memory-Bounded A*

Iterative-Deepening A*

- Like iterative-deepening depth-first, but...
- Depth bound modified to be an **f-limit**
  - Start with f-limit = h(start)
  - Prune any node if f(node) > f-limit
  - Next f-limit=min-cost of any node pruned



Recursive Best-First Search

- Use a variable called f-limit to keep track of the best alternative path available from any ancestor of the current node
- If f(current node) > f-limit, back up to try that alternative path
- As the recursion unwinds, replace the f-value of each node along the path with the backed-up value: the best f-value of its children

Simplified Memory-Bounded A*

- Works like A* until memory is full
- When memory is full, drop the leaf node with the highest f-value (the worst leaf), keeping track of that worst value in the parent
- Complete if any solution is reachable
- Optimal if any optimal solution is reachable
- Otherwise, returns the best reachable solution

Performance of Heuristics

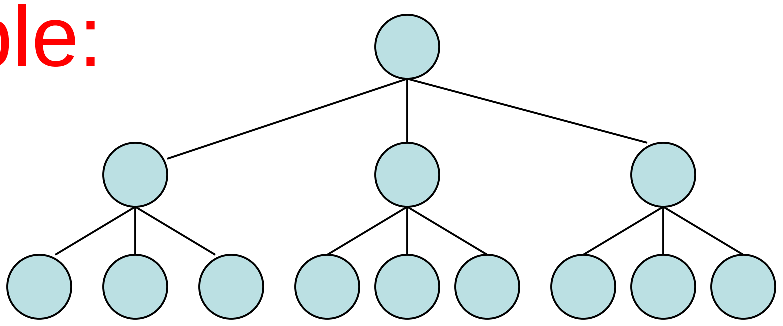
- How do we evaluate a heuristic function?
- effective branching factor b*
  - If A* using h finds a solution at depth d using N nodes, then the effective branching factor is

$$b^* \text{ where } N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- Example:

d=2

b=3



depth 0

depth 1

depth 2

Table of Effective Branching Factors

b	d	N
2	2	7
2	5	63
3	2	13
3	5	364
3	10	88573
6	2	43
6	5	9331
6	10	72,559,411

How might we use this idea to evaluate a heuristic?

How Can Heuristics be Generated?

1. From Relaxed Problems that have fewer constraints but give you ideas for the heuristic function.

2. From Subproblems that are easier to solve and whose exact cost solutions are known.

The cost of solving a relaxed problem or subproblem is not greater than the cost of solving the full problem.

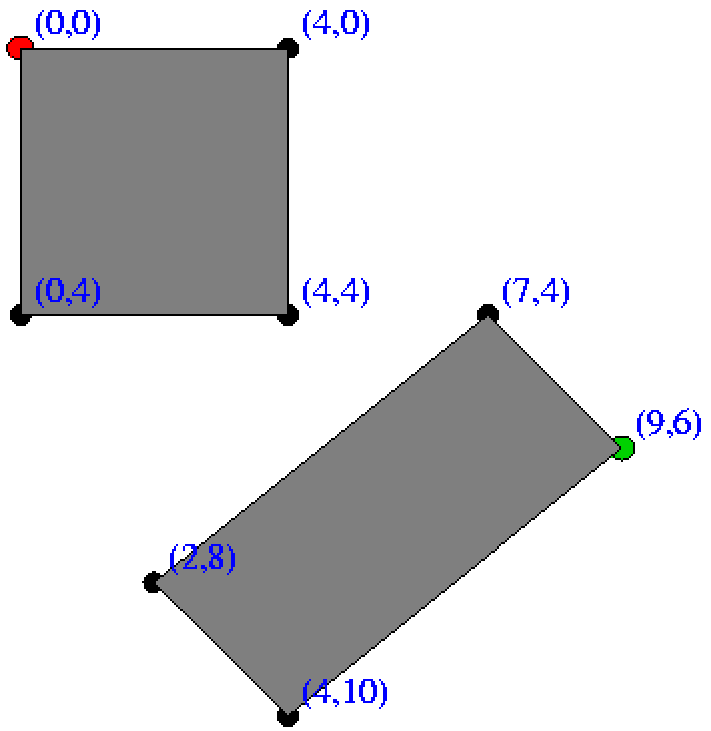
Still may not succeed

- In spite of the use of heuristics and various smart search algorithms, not all problems can be solved.
- Some search spaces are just too big for a classical search.
- So we have to look at other kinds of tools.

HW 2: A* Search

- A robot moves in a 2D space.
- It starts at a start point (x0,y0) and wants to get to a goal point (xg,yg).
- There are rectangular obstacles in the space.
- It cannot go THROUGH the obstacles.
- It can only move to corners of the obstacles, ie. search space limited.

Simple Data Set



How can the robot get from (0,0) to (9,6)?
What is the minimal length path?

More next time.