Chapter 04

Informed Search

Instructor

LE Thanh Sach, Ph.D.

Instructor's Information

LE Thanh Sach, Ph.D.

Office:

Department of Computer Science,

Faculty of Computer Science and Engineering,

HoChiMinh City University of Technology.

Office Address:

268 LyThuongKiet Str., Dist. 10, HoChiMinh City, Vietnam.

E-mail: LTSACH@cse.hcmut.edu.vn

E-home: http://cse.hcmut.edu.vn/~ltsach/

Tel: (+84) 83-864-7256 (Ext: 5839)

Acknowledgment

The slides in this PPT file are composed using the materials supplied by

- **Prof. Stuart Russell and Peter Norvig**: They are currently from University of California, Berkeley. They are also the author of the book "Artificial Intelligence: A Modern Approach", which is used as the textbook for the course
- **Prof. Tom Lenaerts**, from Université Libre de Bruxelles

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*)

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

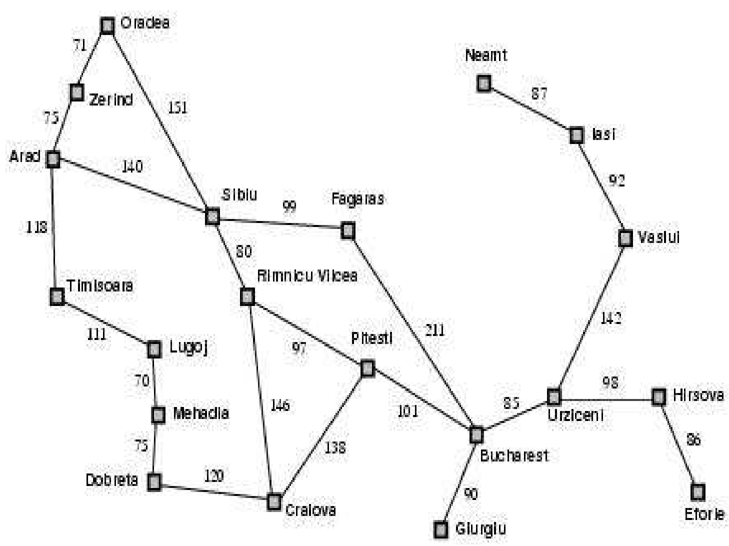
A heuristic function

- [dictionary]*"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

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

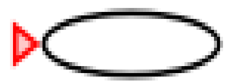


- h_{SLD}=straight-line distance heuristic.
- h_{SLD} can **NOT** be computed from the problem description itself
- In this example $f(n)=h(n)$
  - Expand node that is closest to goal

= Greedy best-first search

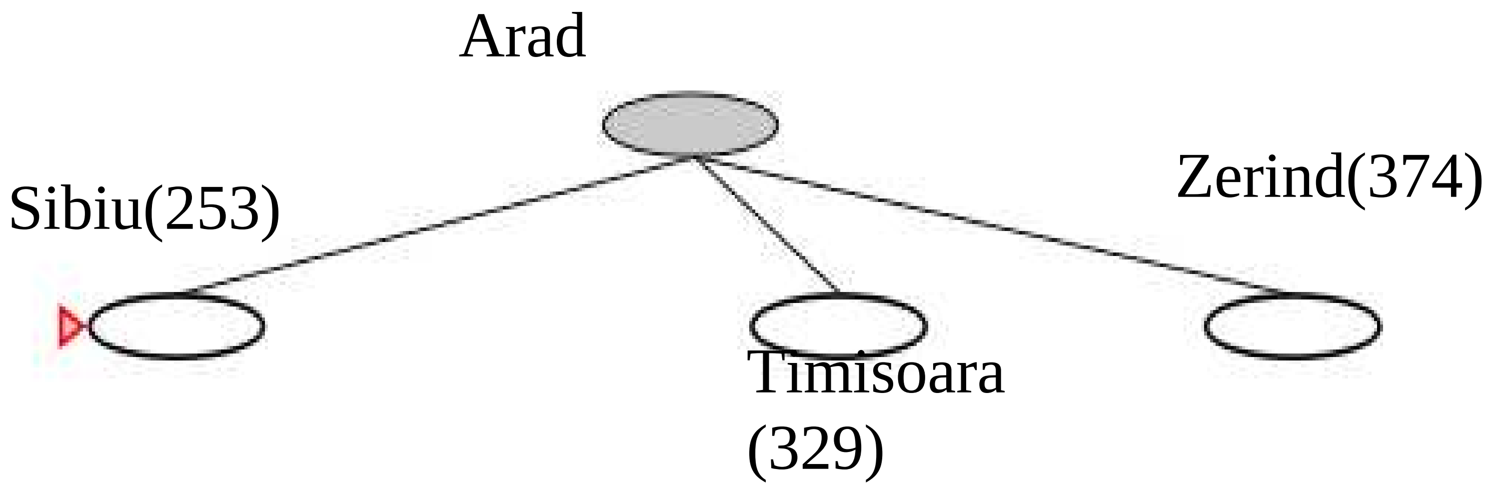
Greedy search example

Arad (366)



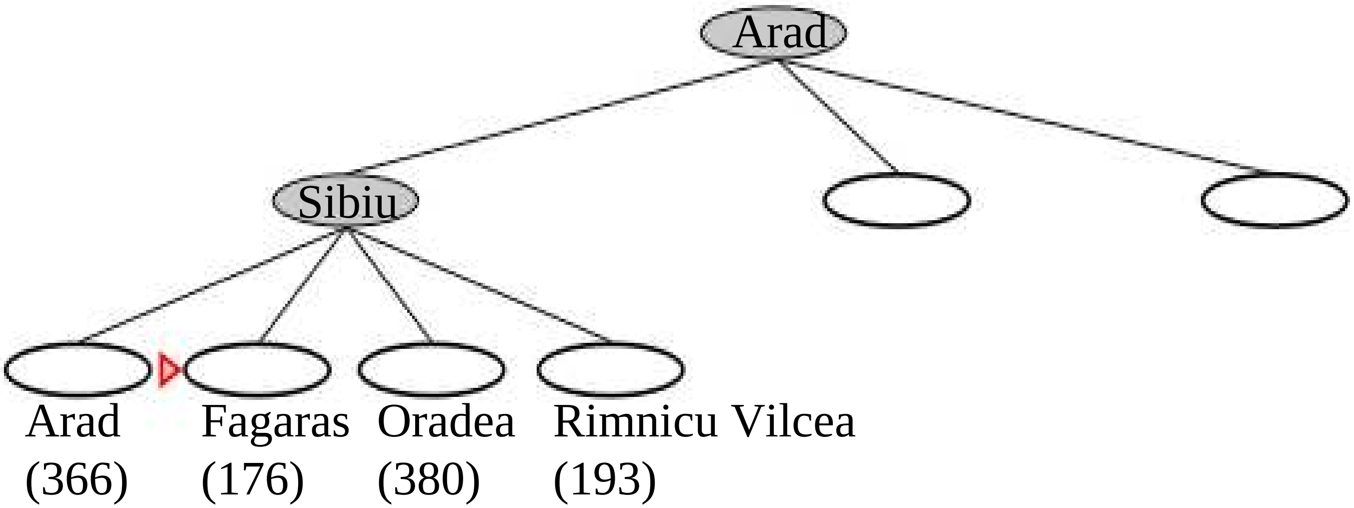
- Assume that we want to use greedy search to solve the problem of travelling from Arad to Bucharest.
- The initial state=Arad

Greedy search example



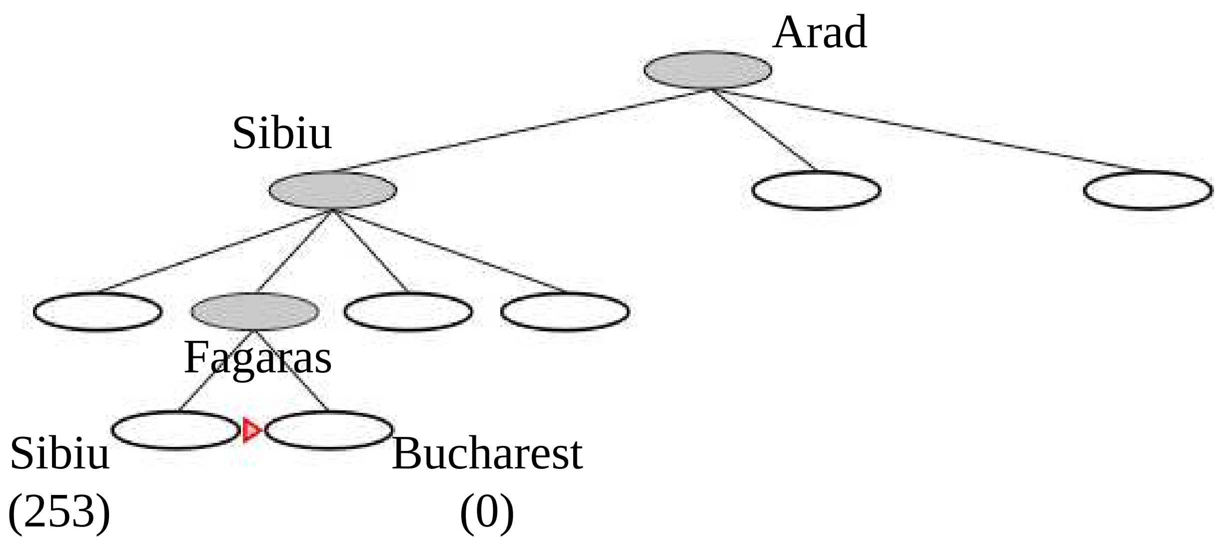
- ❖ The first expansion step produces:
  - Sibiu, Timisoara and Zerind
- ❖ Greedy best-first will select Sibiu.

Greedy search example



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

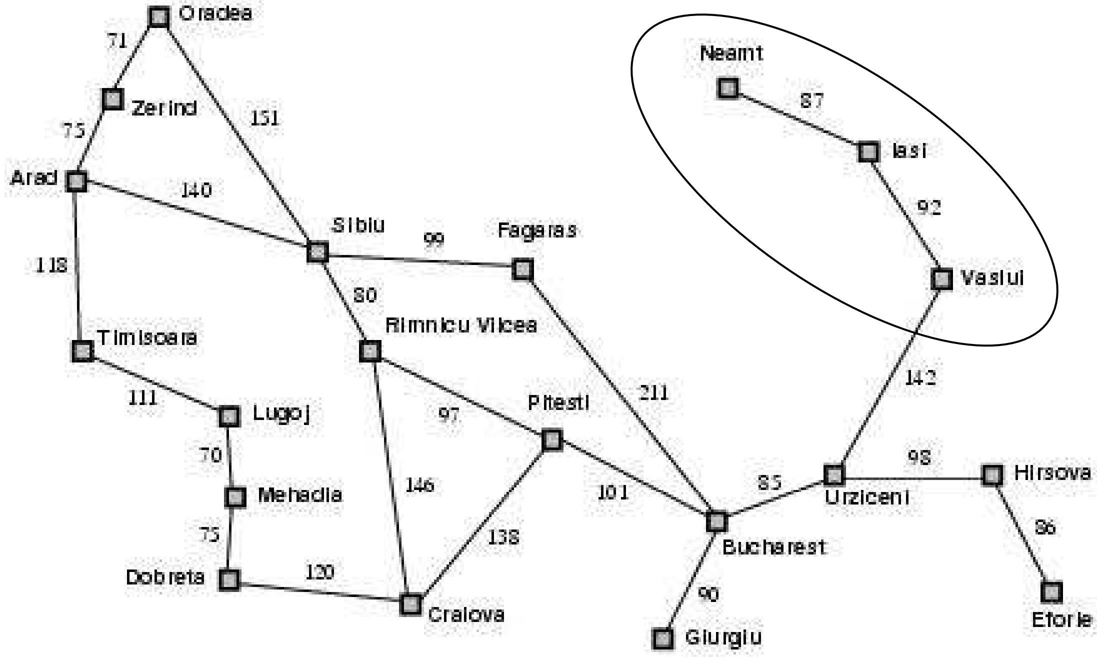
Greedy search example



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

Greedy search, evaluation

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



Greedy search, evaluation

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

Greedy search, evaluation

- Completeness: NO (cfr. DF-search)
- Time complexity: $O(b^m)$
- Space complexity: $O(b^m)$
- Optimality? NO
  - Same as DF-search

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

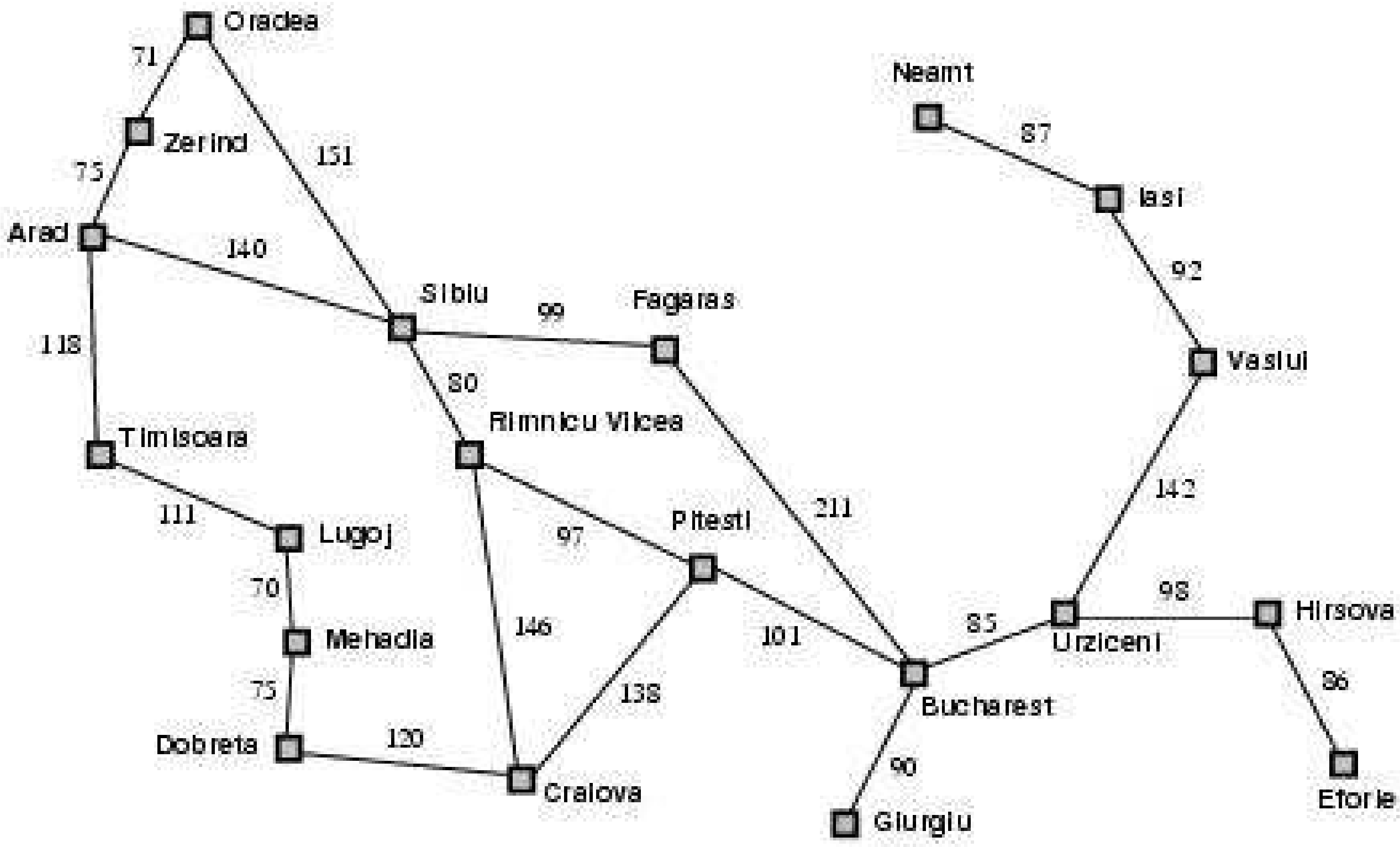
- A* search uses an admissible heuristic
  - A heuristic is admissible if it *never overestimates* the cost to reach the goal
  - Are optimistic

Formally:

1. $h(n) <= h^*(n)$ where $h^*(n)$ is the true cost from n
2. $h(n) >= 0$ so $h(G)=0$ for any goal G.

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

Romania example

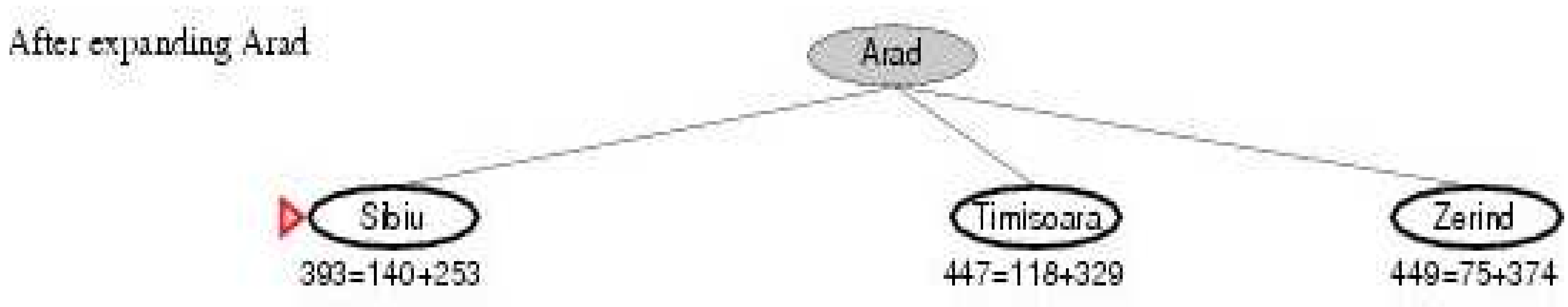


A* search example

(a) The initial state

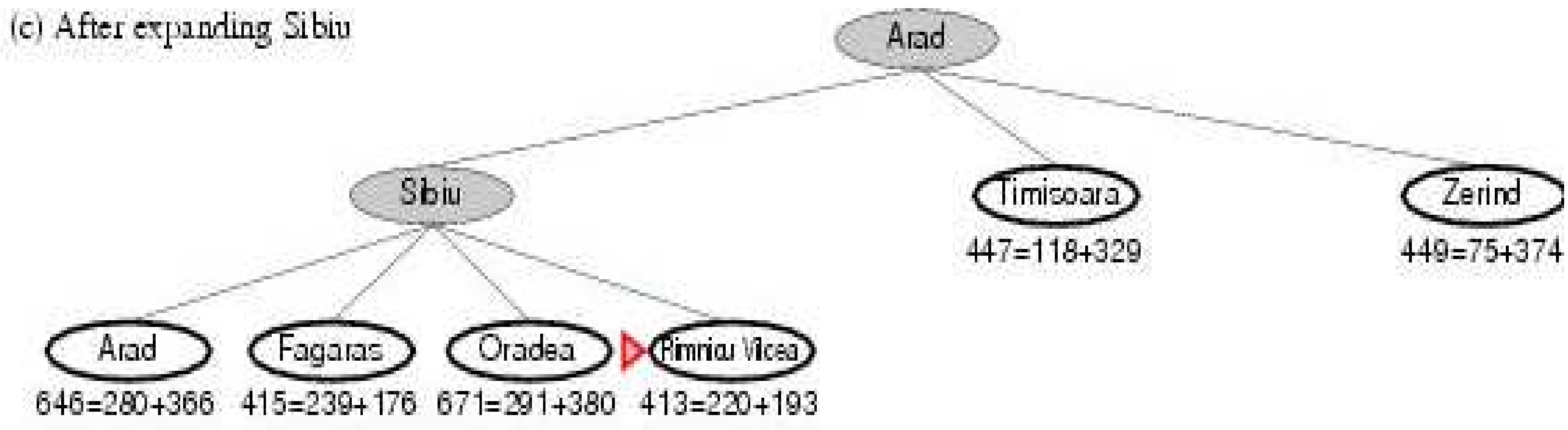
- Find Bucharest starting at Arad
  - f(Arad) = c(??,Arad)+h(Arad)=0+366=366

A* search example



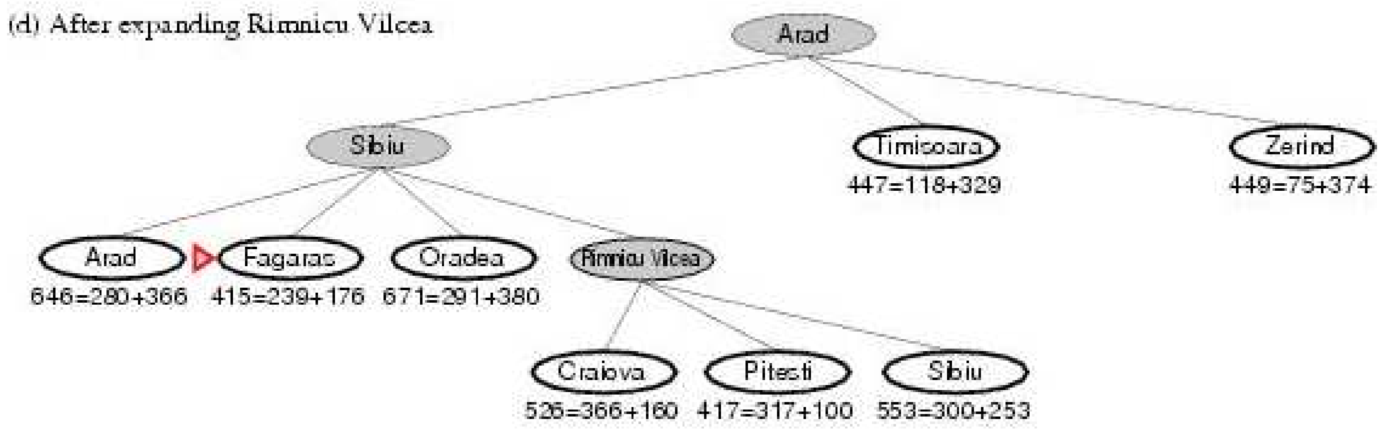
- Expand Arrad and determine *f(n)* for each node
  - f(Sibiu)=c(Arad,Sibiu)+h(Sibiu)=140+253=393
  - f(Timisoara)=c(Arad,Timisoara)+h(Timisoara)=118+329=447
  - f(Zerind)=c(Arad,Zerind)+h(Zerind)=75+374=449
- Best choice is Sibiu

A* search example



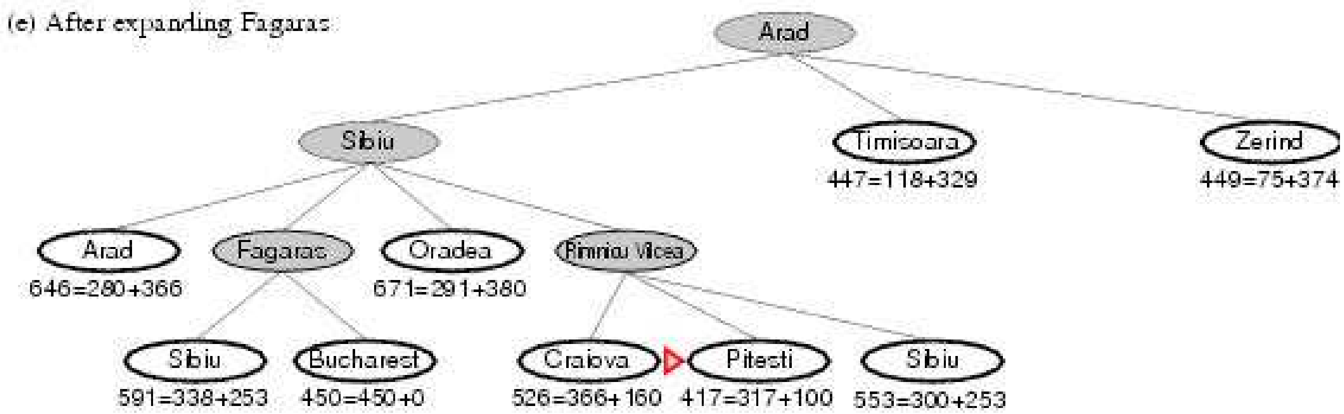
- Expand Sibiu and determine *f(n)* for each node
  - f(Arad)=c(Sibiu,Arad)+h(Arad)=280+366=646
  - f(Fagaras)=c(Sibiu,Fagaras)+h(Fagaras)=239+179=415
  - f(Oradea)=c(Sibiu,Oradea)+h(Oradea)=291+380=671
  - f(Rimnicu Vilcea)=c(Sibiu,Rimnicu Vilcea)+
    h(Rimnicu Vilcea)=220+192=413
- Best choice is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

A* search example



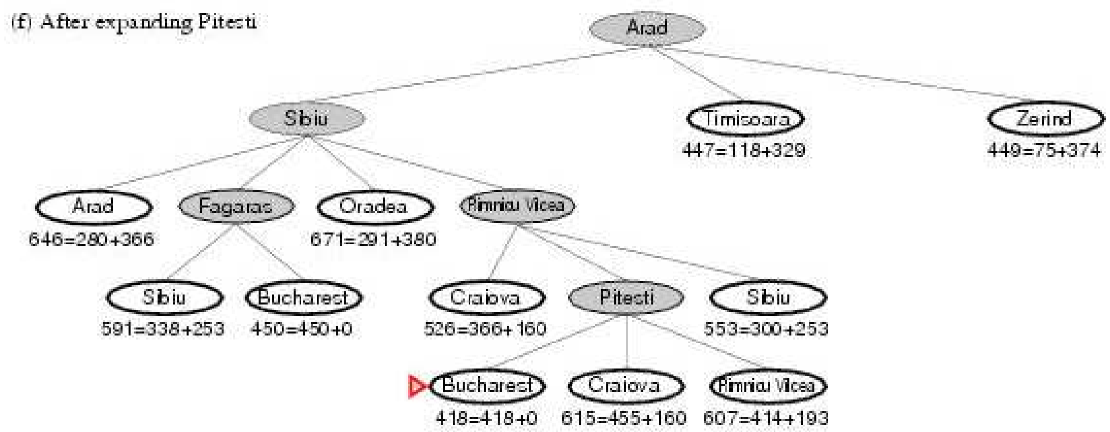
- Expand Rimnicu Vilcea and determine *f(n)* for each node
  - f(Craiova)=c(Rimnicu Vilcea, Craiova)+h(Craiova)=360+160=526
  - f(Pitesti)=c(Rimnicu Vilcea, Pitesti)+h(Pitesti)=317+100=417
  - f(Sibiu)=c(Rimnicu Vilcea,Sibiu)+h(Sibiu)=300+253=553
- Best choice is Fagaras

A* search example



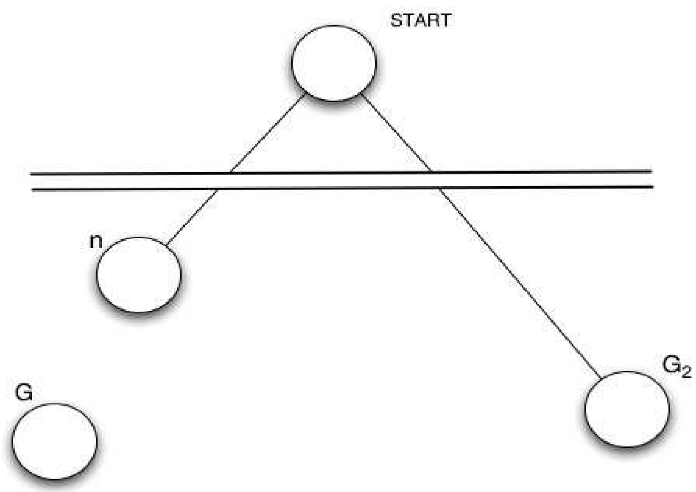
- Expand Fagaras and determine *f(n)* for each node
  - f(Sibiu)=c(Fagaras, Sibiu)+h(Sibiu)=338+253=591
  - *f(Bucharest)=c(Fagaras,Bucharest)+h(Bucharest)=450+0=450*
- Best choice is Pitesti !!!

A* search example



- Expand Pitesti and determine *f(n)* for each node
  - *f(Bucharest)=c(Pitesti,Bucharest)+h(Bucharest)=418+0=418*
- Best choice is Bucharest !!!
  - Optimal solution (only if *h(n)* is admissable)
- Note values along optimal path !!

Optimality of A*(standard proof)



- Suppose suboptimal goal G_2 in the queue.
- Let n be an unexpanded node on a shortest to optimal goal G.

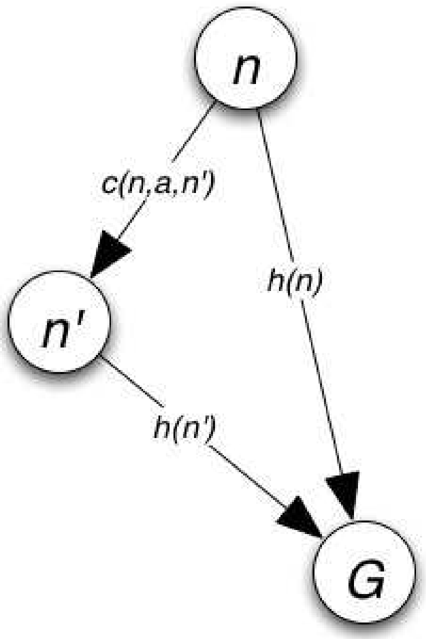
$f(G_2)$	$= g(G_2)$	since $h(G_2)=0$
	$> g(G)$	since G_2 is suboptimal
	$>= f(n)$	since h is admissible

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

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

$$h(n) \le c(n,a,n') + h(n')$$

- If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) \\ &\ge f(n) \end{aligned}$$

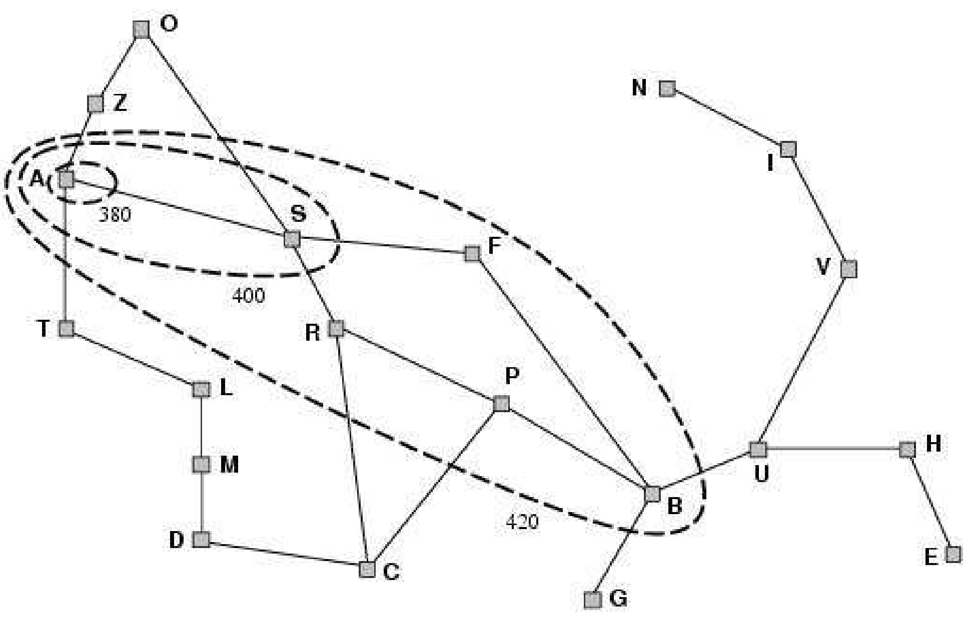
i.e. f(n) is nondecreasing along any path.

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 $f=f_i$, where $f_i < f_i+1$.



A* search, evaluation

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

A* search, evaluation

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

A* search, evaluation

- Completeness: YES
- Time complexity: (exponential with path length)
- 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+1} until f_i 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)

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

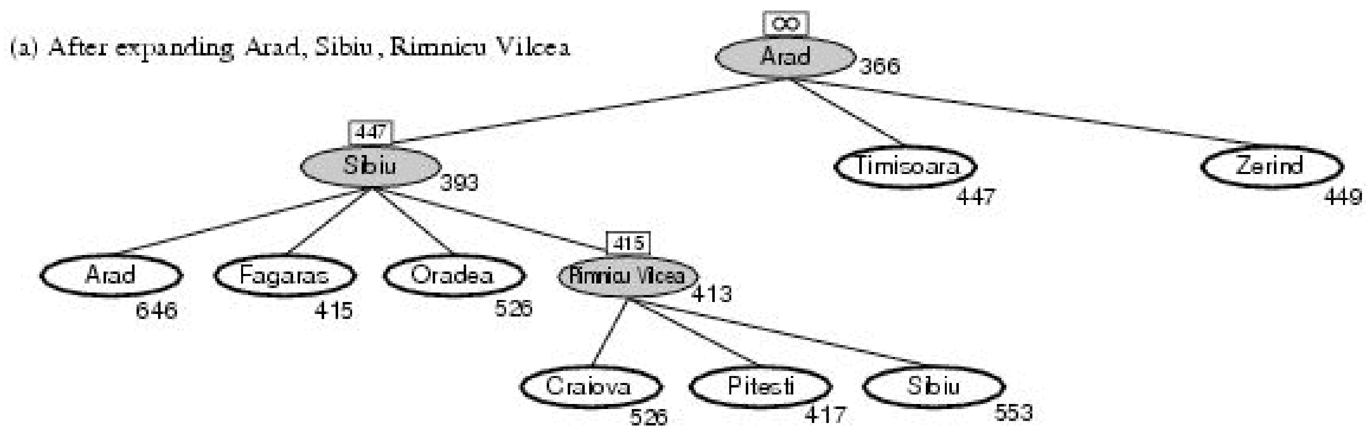
function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **return** a solution or failure
  return RFBS(*problem*,MAKE-NODE(INITIAL-STATE[*problem*]),∞)

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, ∞
  for each *s* **in** *successors* **do**
    f [*s*] ← 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*] ← RBFS*(problem, best,* min(*f_limit, alternative*))
    if *result* ≠ failure **then return** *result*

Recursive best-first search

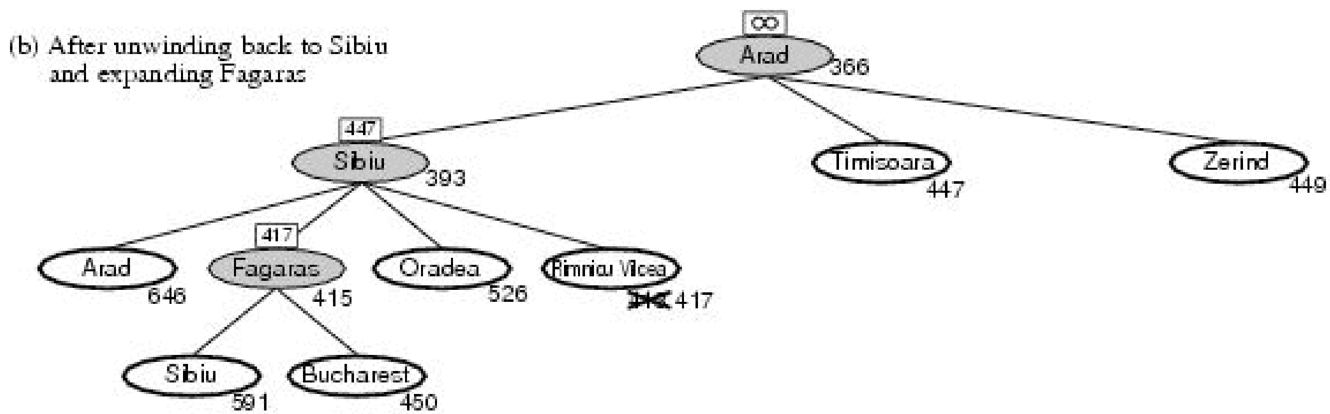
- Keeps track of the f-value of the best-alternative path available.
  - If current f-values exceeds this alternative f-value than backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
  - Re-expansion of this result is thus still possible.

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*.

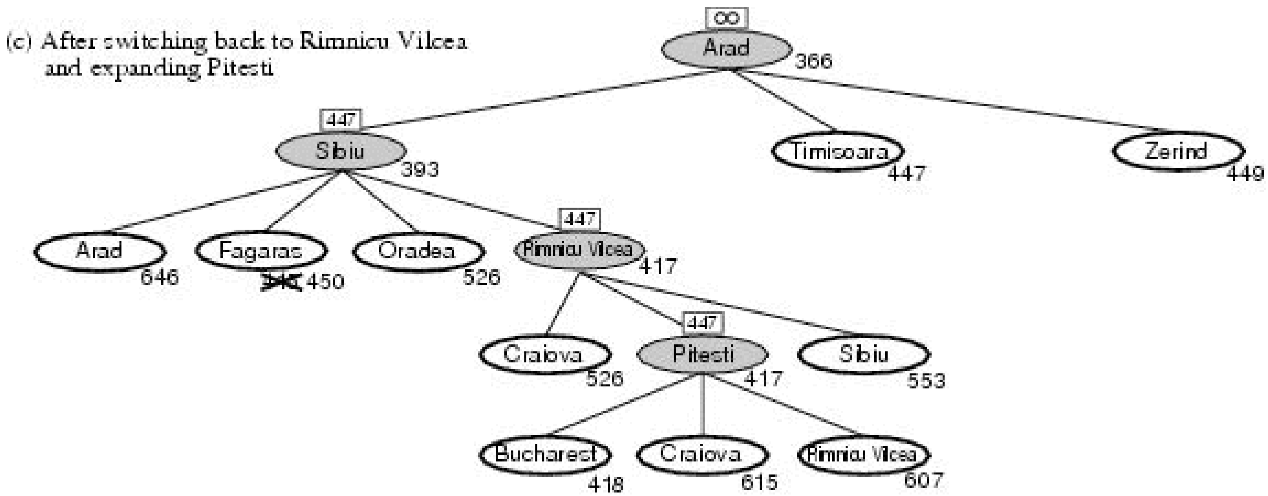
Recursive best-first search, ex.



- 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 RBFS 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, *f* [*best*] ← RBFS(*problem*, *best*, min(*f_limit*, *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 because 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)*.
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
- IDA* en RBFS suffer from ***too little*** memory.

(simplified) memory-bounded A*

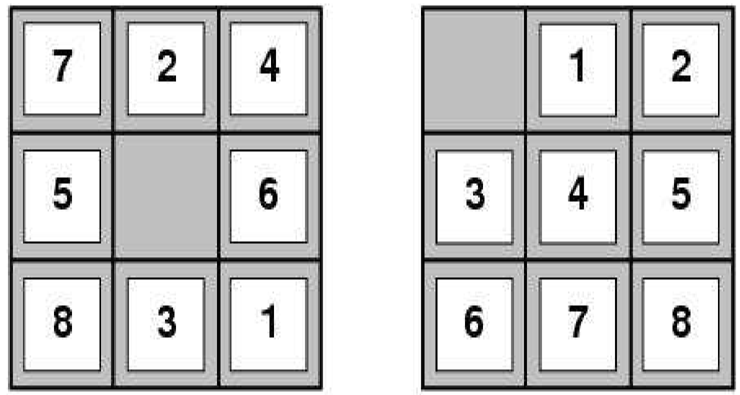
- Use all available memory.
  - I.e. expand best leafs until available memory is full
  - When full, SMA* drops worst leaf node (highest *f*-value)
  - Like RFBS backup forgotten node to its parent

- 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.

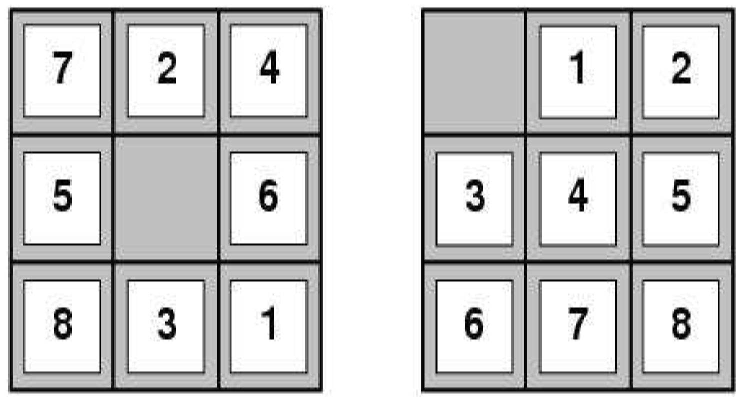
Heuristic functions



Start State Goal State

- E.g for the 8-puzzle
  - Avg. solution cost is about 22 steps (branching factor +/- 3)
  - Exhaustive search to depth 22: 3.1 x 10^{10} states.
  - A good heuristic function can reduce the search process.

Heuristic functions



Start State Goal State

- E.g for the 8-puzzle knows two commonly used heuristics
- h_1 = the number of misplaced tiles
  - $h_1(s)=8$
- h_2 = 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.

Heuristic quality and dominance

❖ 1200 random problems with solution lengths from 2 to 24.

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

❖ If $h_2(n) >= h_1(n)$ for all n (both admissible)

then h_2 *dominates* h_1 and is better for search

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.

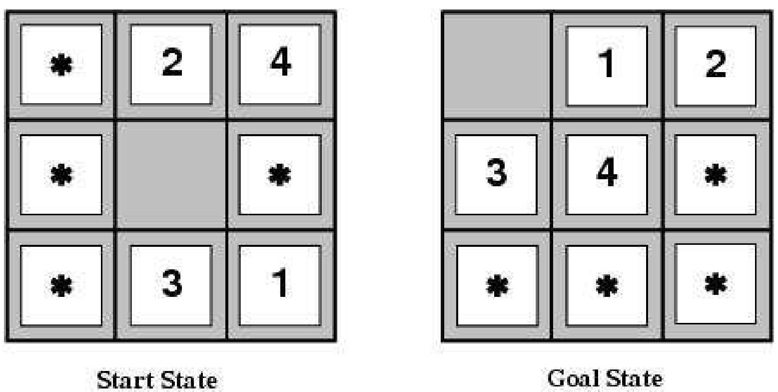
    As a result, $h_2(n)$ gives the shortest solution.

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

ABSolver found a usefull heuristic for the rubic 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 for every possible subproblem instance.
  - The complete heuristic is constructed using the patterns in the DB

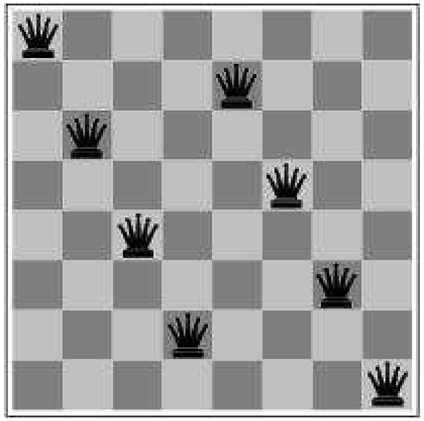


Inventing admissible heuristics

- 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

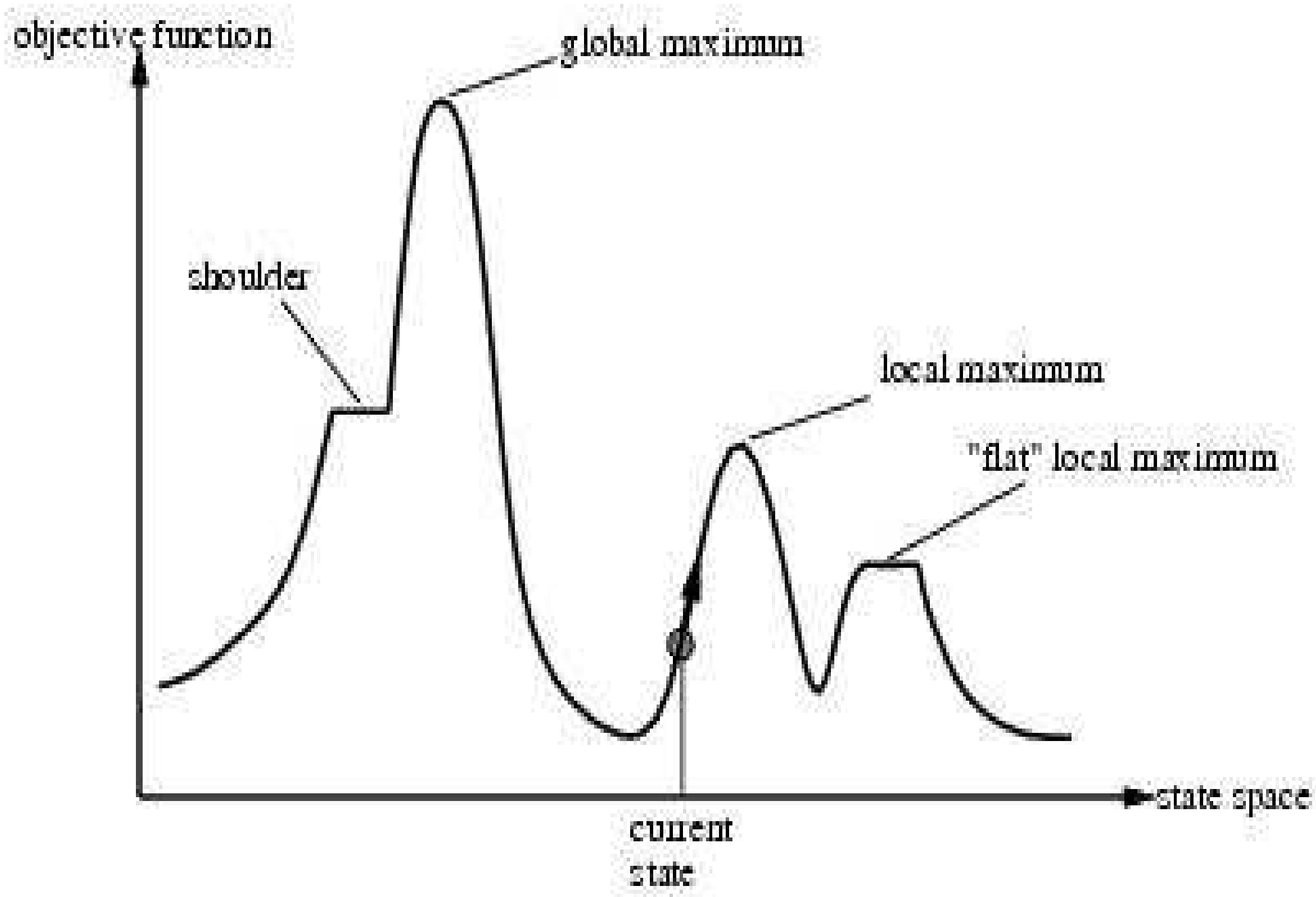


- Different algorithms can be used
  - Local search

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.

Local search and 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 search

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*

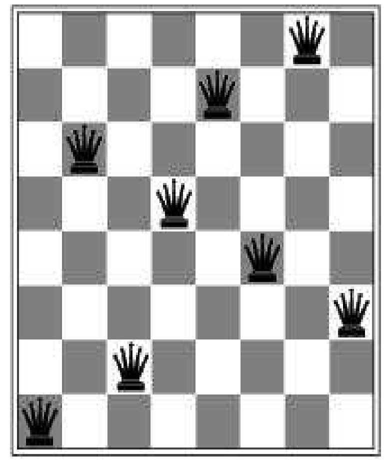
Hill-climbing example

- 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 example

a)

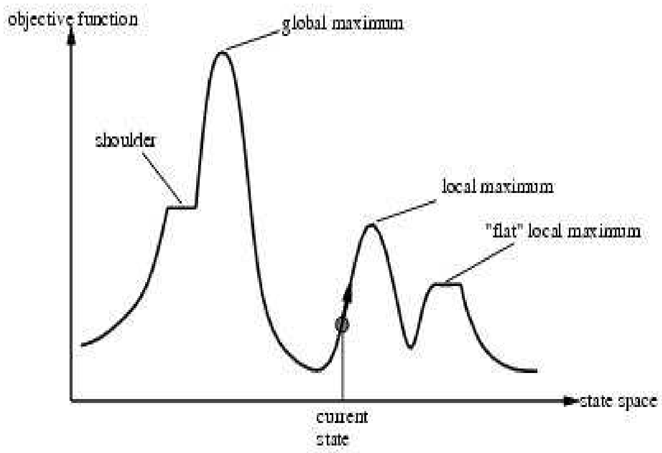
b)

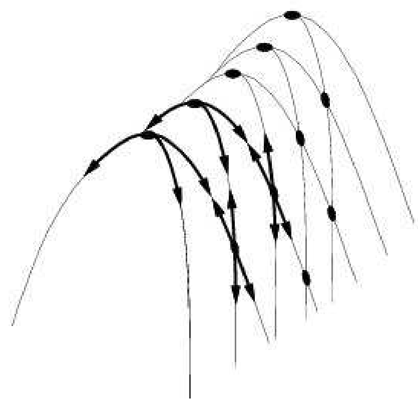


a) shows a state of h=17 and the h-value for each possible successor.

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.

Simulated annealing

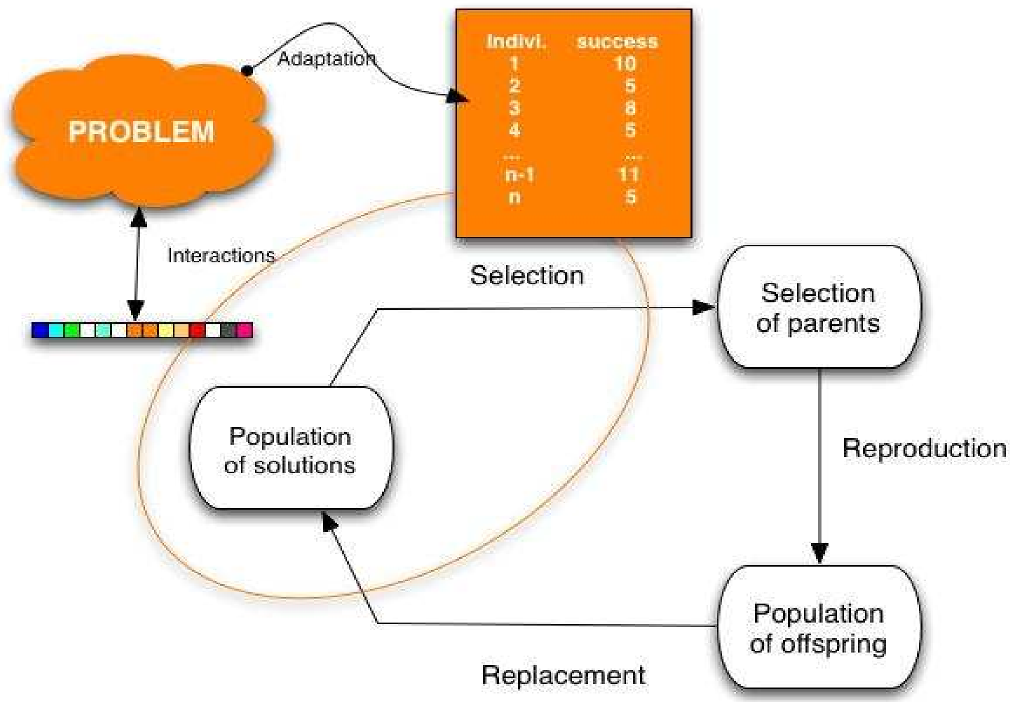
```
function SIMULATED-ANNEALING( problem, schedule) return a solution state
    input: problem, a problem
           schedule, a mapping from time to temperature
    local variables: current, a node.
                     next, a node.
                     T, a "temperature" controlling the probability of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ←  VALUE[next] - VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(ΔE /T)
```

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 $\rightarrow$ 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.

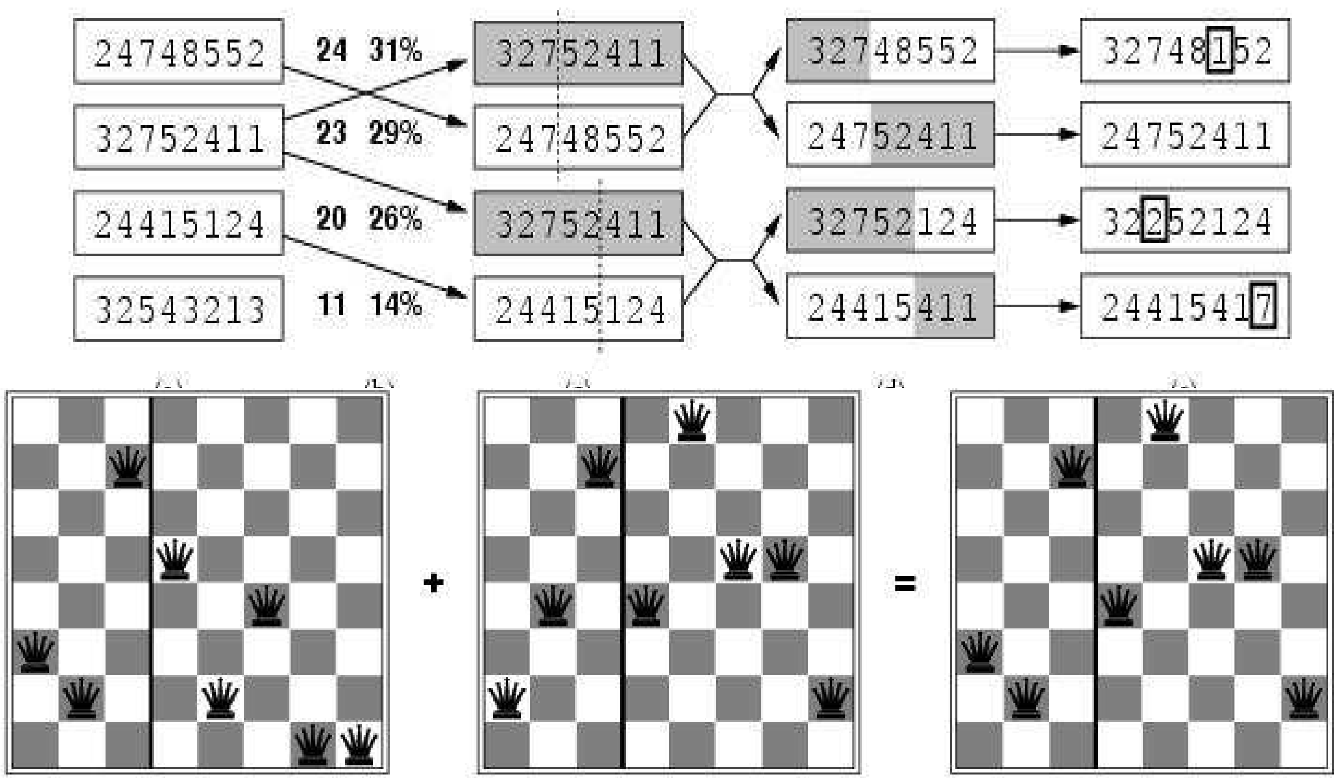
Genetic algorithms

❖ Variant of local beam search with *sexual recombination.*



Genetic algorithms

❖ Variant of local beam search with *sexual recombination.*



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

Supplementary Documents for Hill Climbing

Instructor

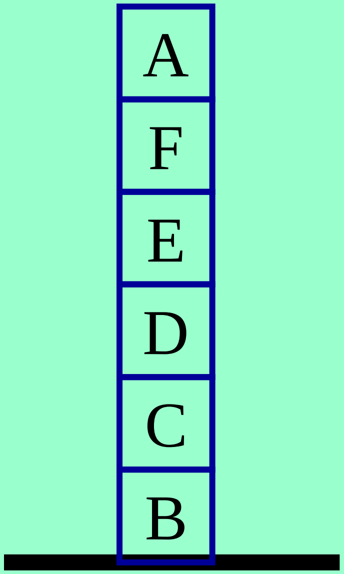
LE Thanh Sach, Ph.D.

Outline

- Block World
- N-Puzzle
- N-Queens

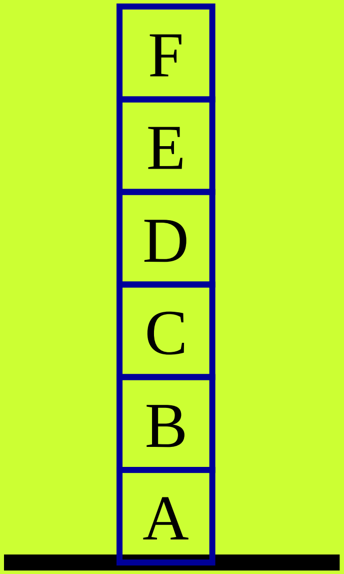
Block World - Heuristics

- Local heuristic, referred to as H1:
  - For ever block that is resting on the right thing (another block or table) compared to goal state:
    - ✓ +1 point
  - For ever block that is resting on the wrong thing (another block or table) compared to goal state:
    - ✓ -1 point



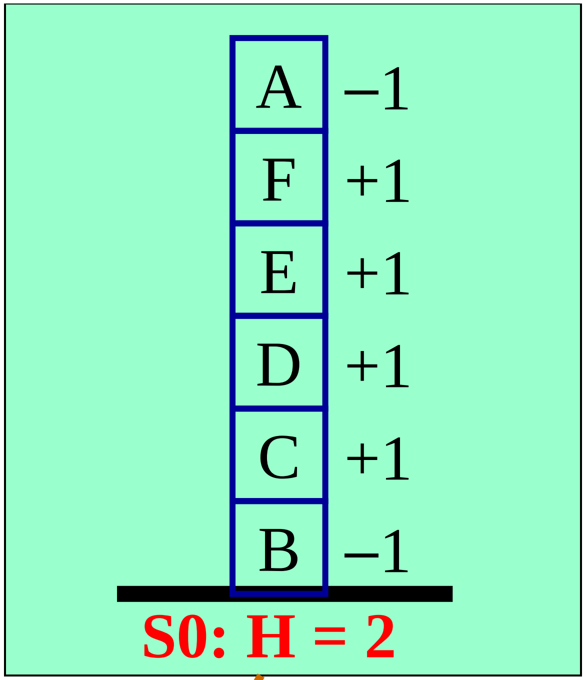
Initial State

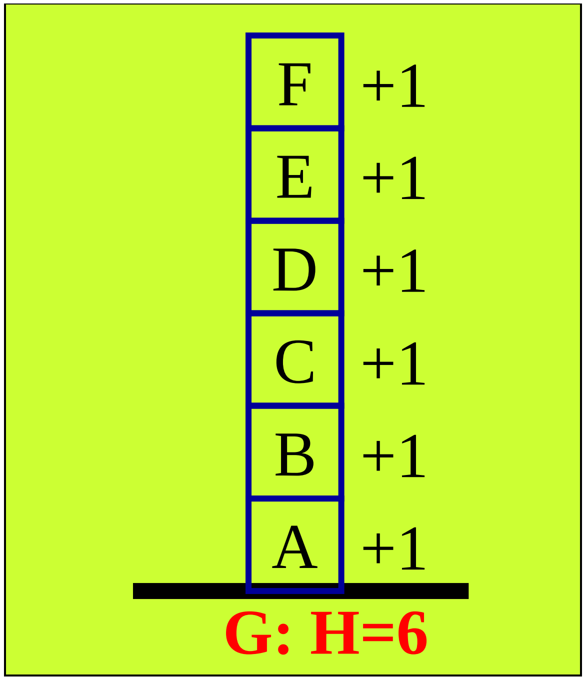
(S0)

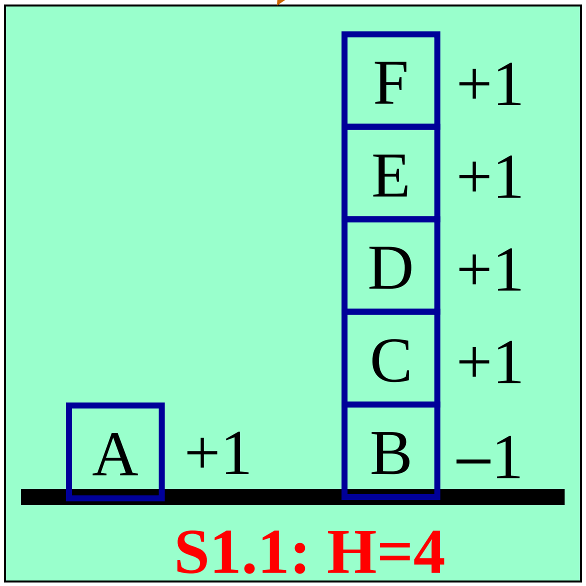


Goal State

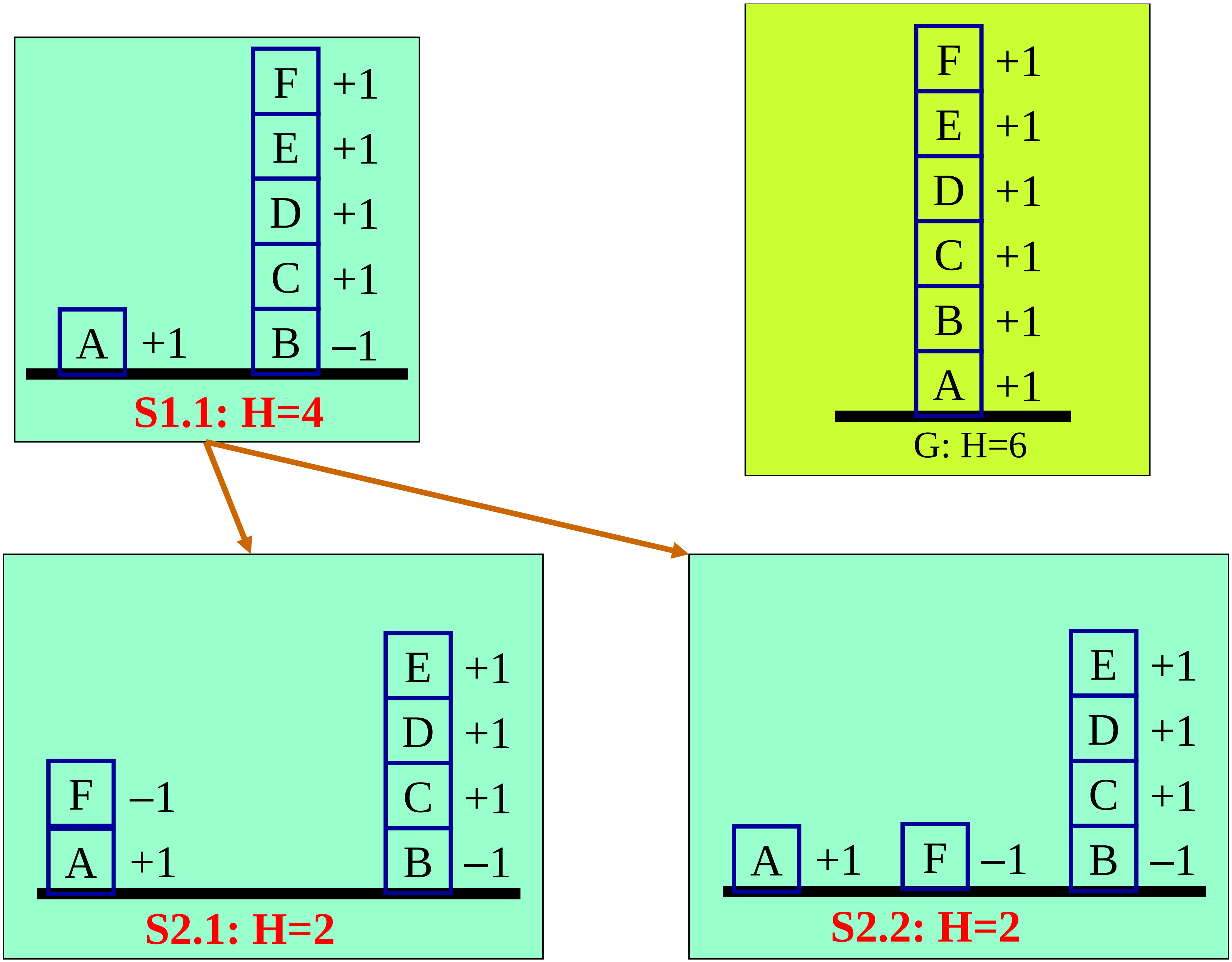
(G)

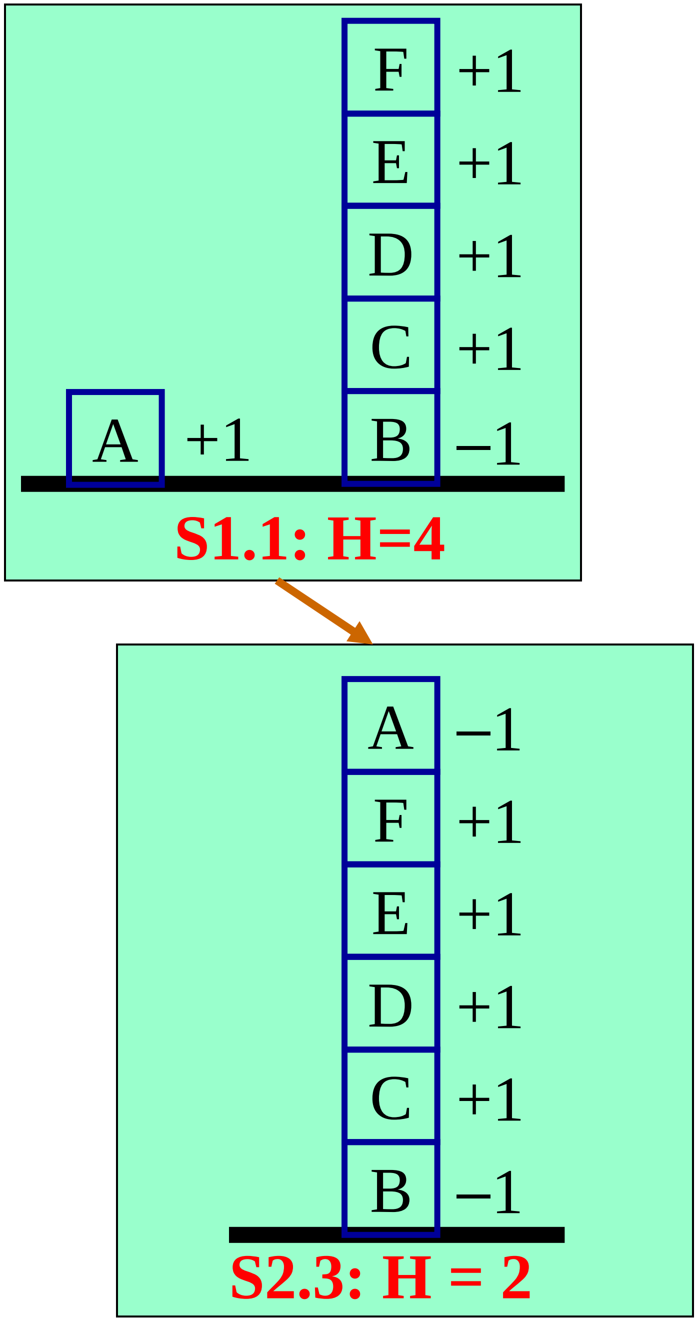


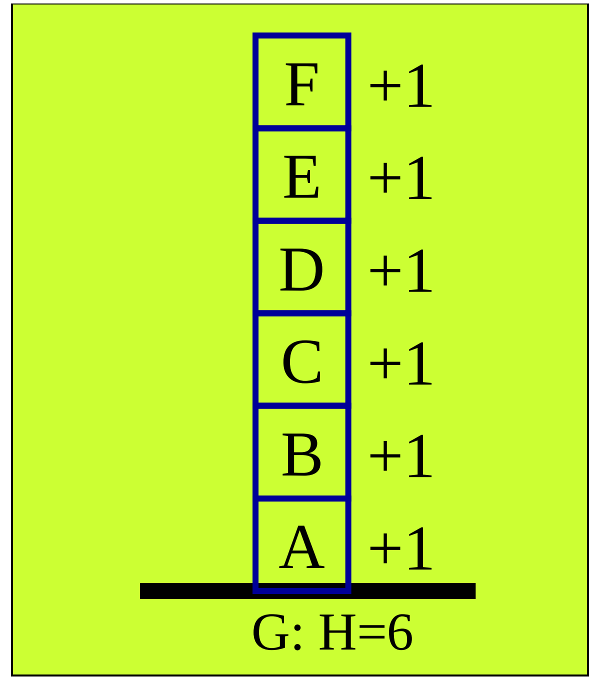




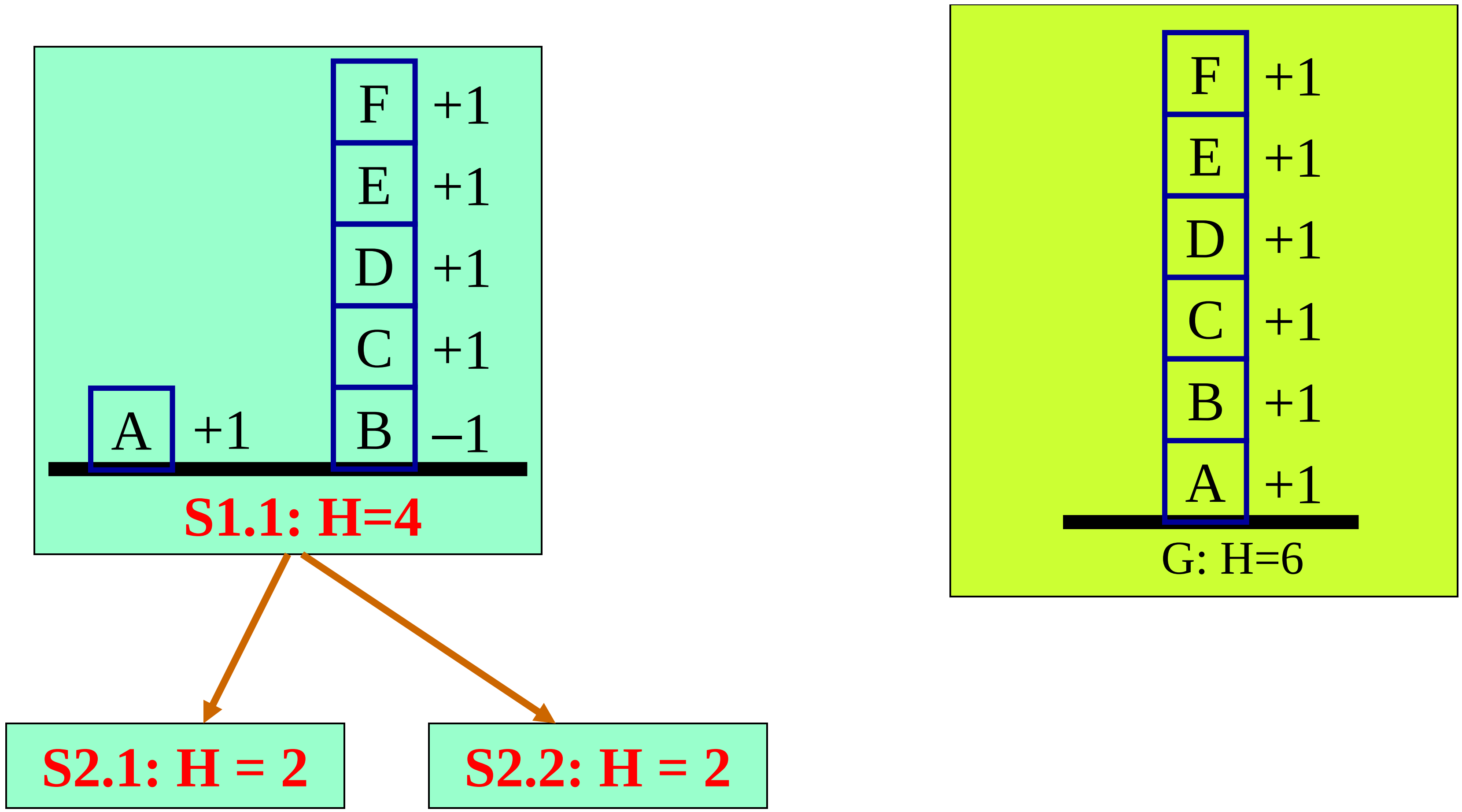
S1.1 is better than S0
→ S0 become current state


F +1
E +1
D +1
C +1
A +1
B –1
S1.1: H=4
F +1
E +1
D +1
C +1
B +1
A +1
G: H=6
E +1
D +1
F –1
C +1
A +1
B –1
S2.1: H=2
E +1
D +1
C +1
A +1
F –1
B –1
S2.2: H=2





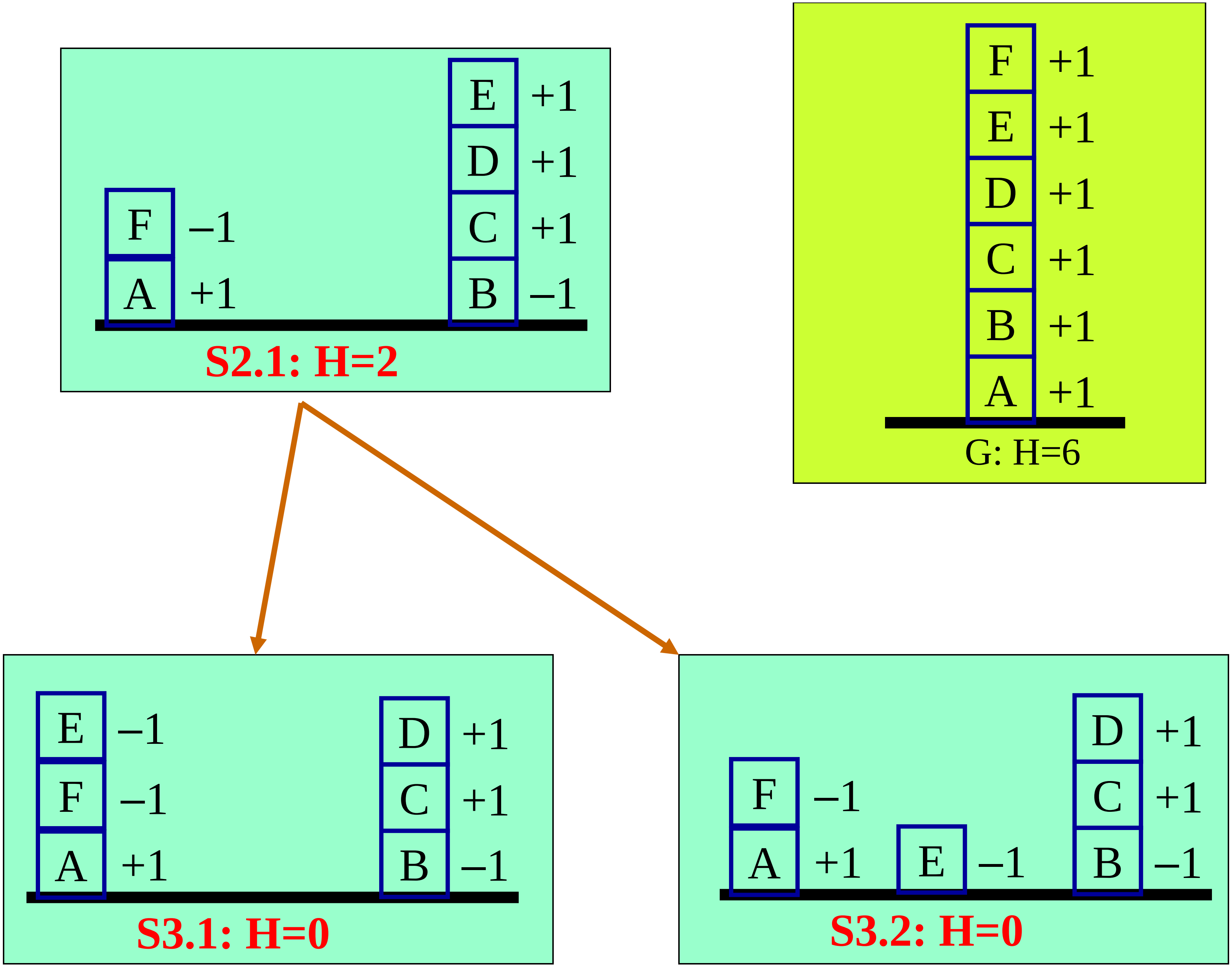
(Repeated State, removed)

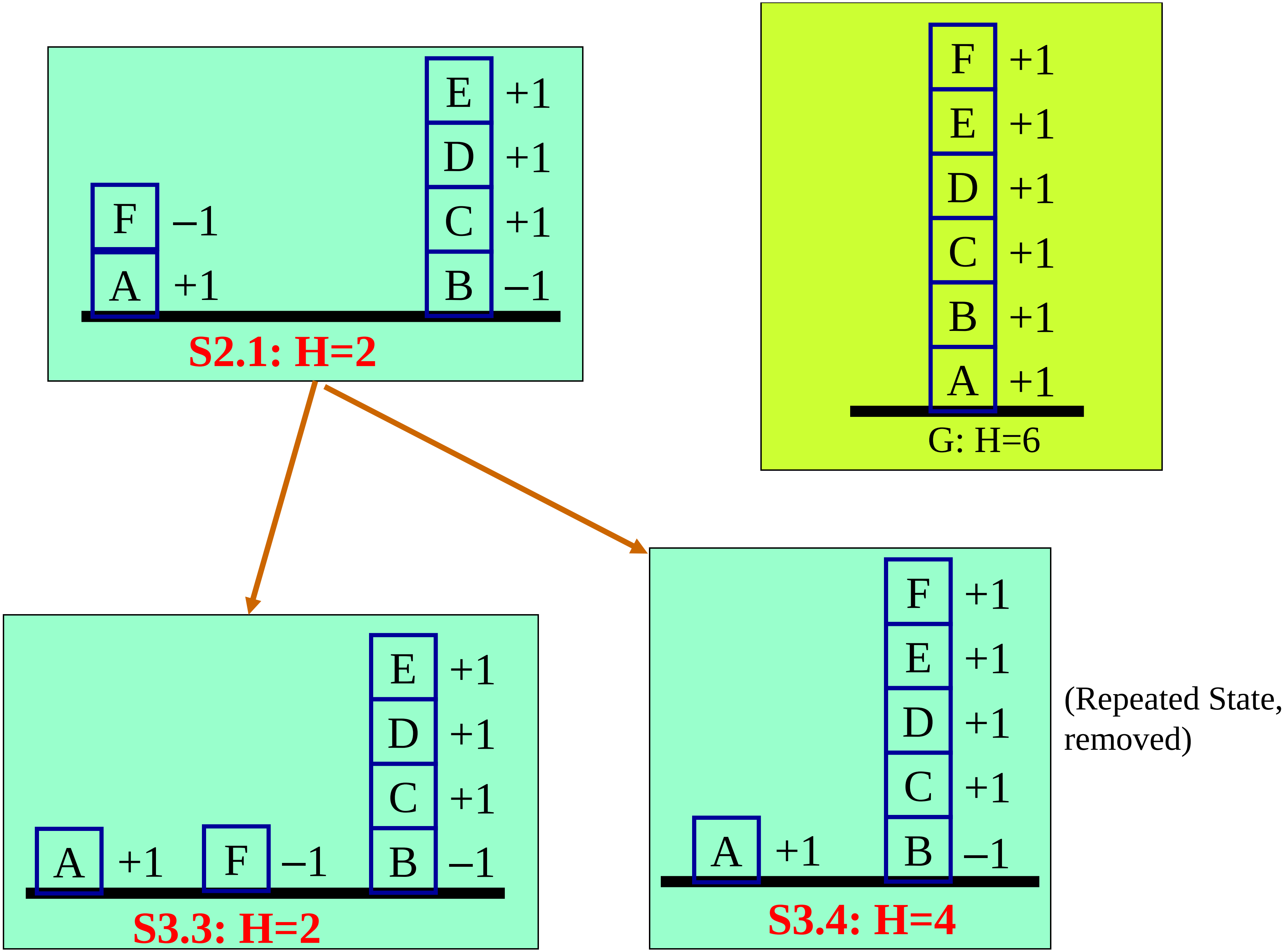


Both of S2.1 and S2.2 have the same score

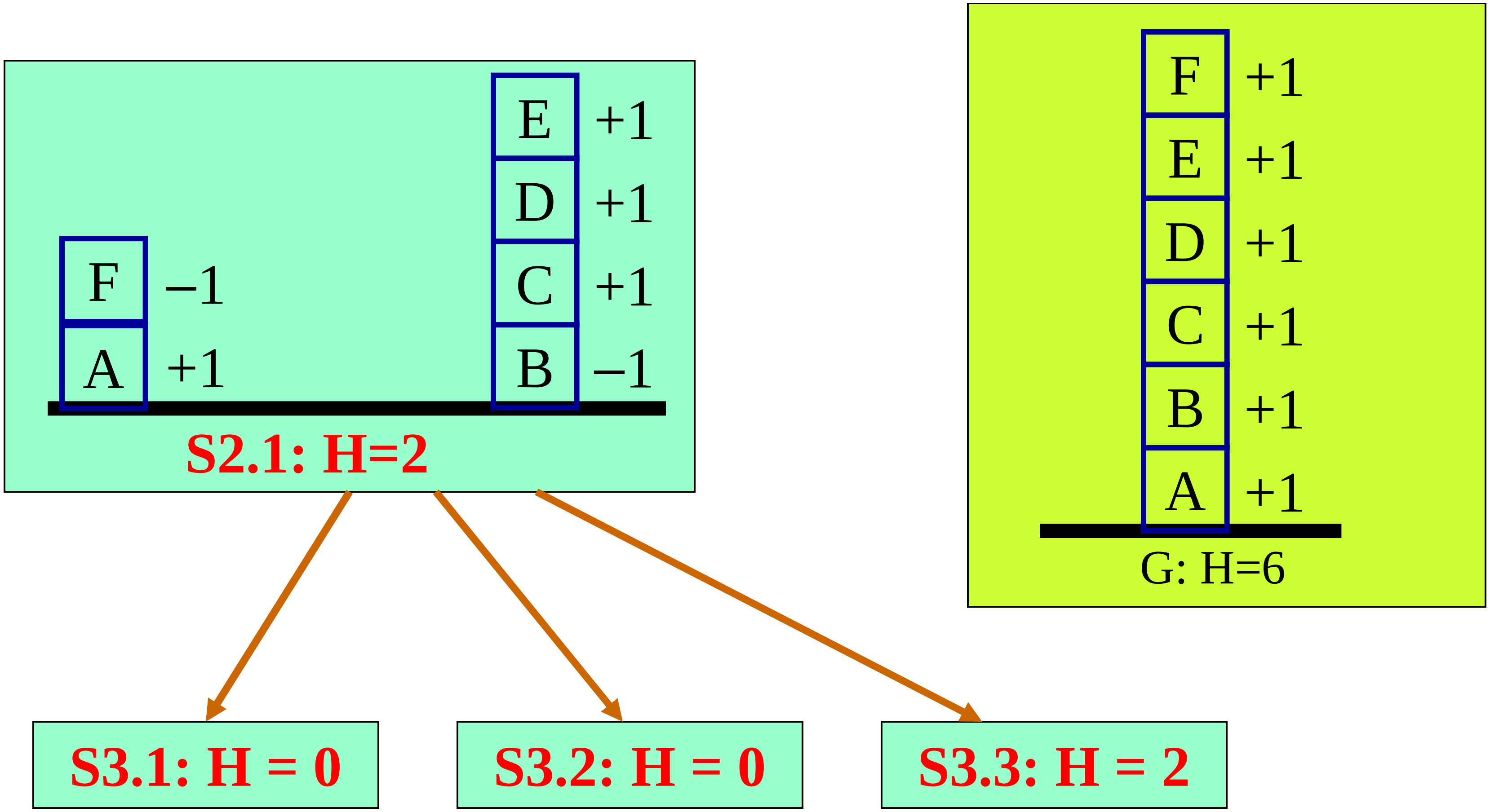
→Select any of them

→Suppose S2.1 is selected to become the current state


E +1
D +1
F –1
C +1
A +1
B –1
S2.1: H=2
F +1
E +1
D +1
C +1
B +1
A +1
G: H=6
E –1
D +1
F –1
C +1
A +1
B –1
S3.1: H=0
D +1
F –1
C +1
A +1
E –1
B –1
S3.2: H=0



F –1
A +1
E +1
D +1
C +1
B –1
S2.1: H=2
F +1
E +1
D +1
C +1
B +1
A +1
G: H=6
E +1
D +1
C +1
A +1
F –1
B –1
S3.3: H=2
F +1
E +1
D +1
C +1
A +1
B –1
S3.4: H=4
(Repeated State, removed)

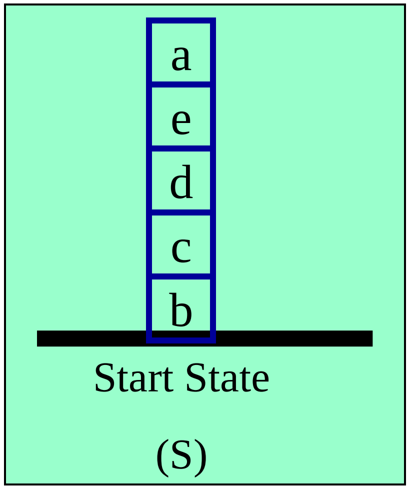


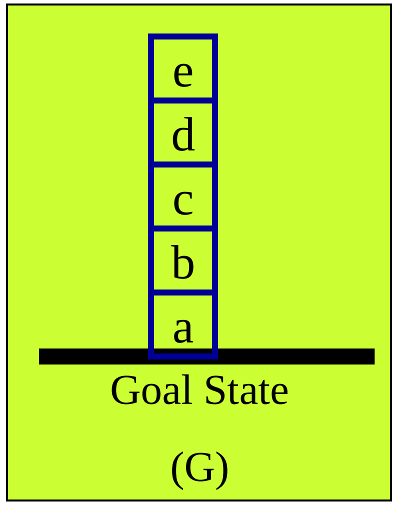
None of S3.1, S3.2, and S3.3 is better than their father

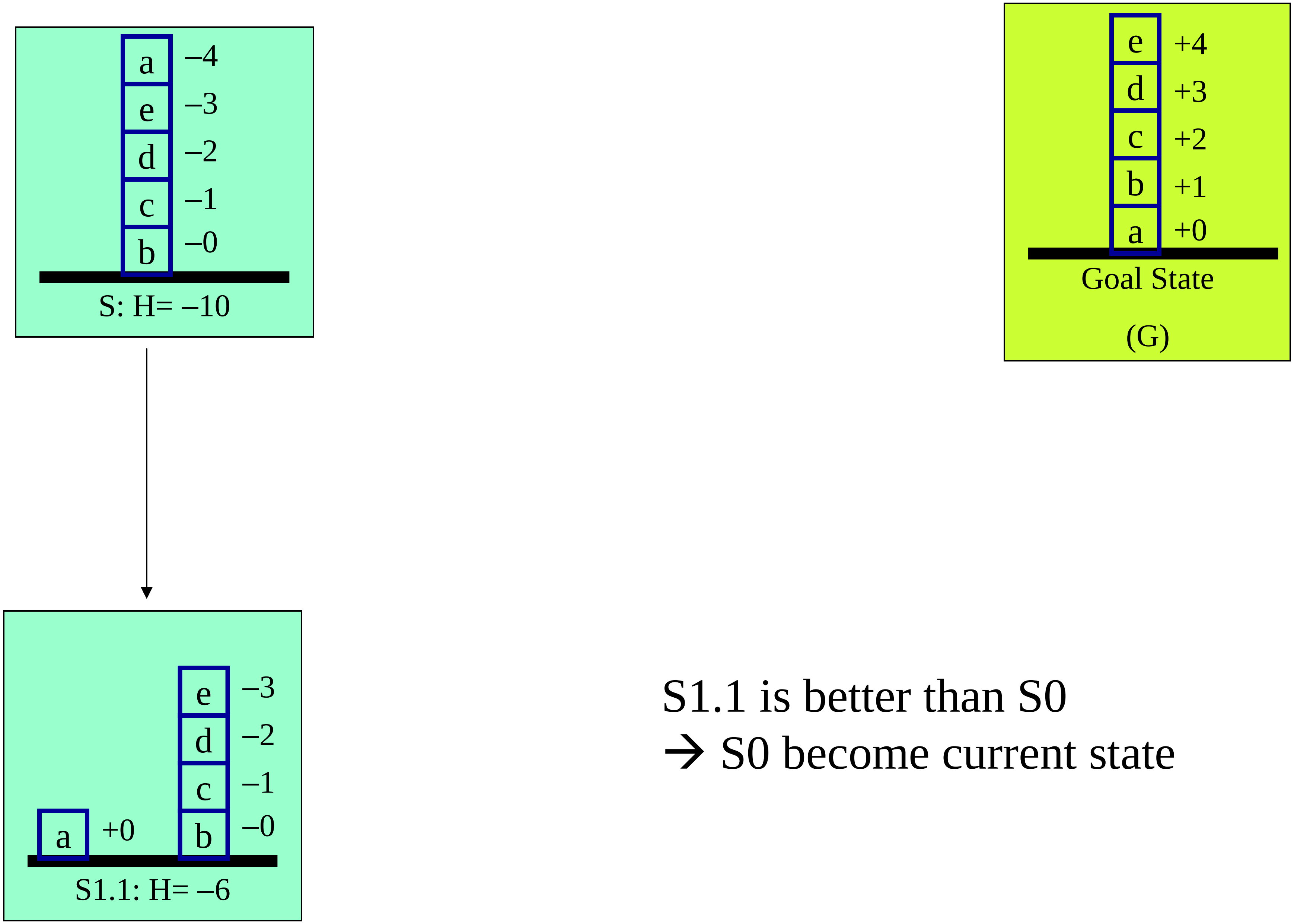
→ STOP Hill-Climbing

Block World - Heuristics

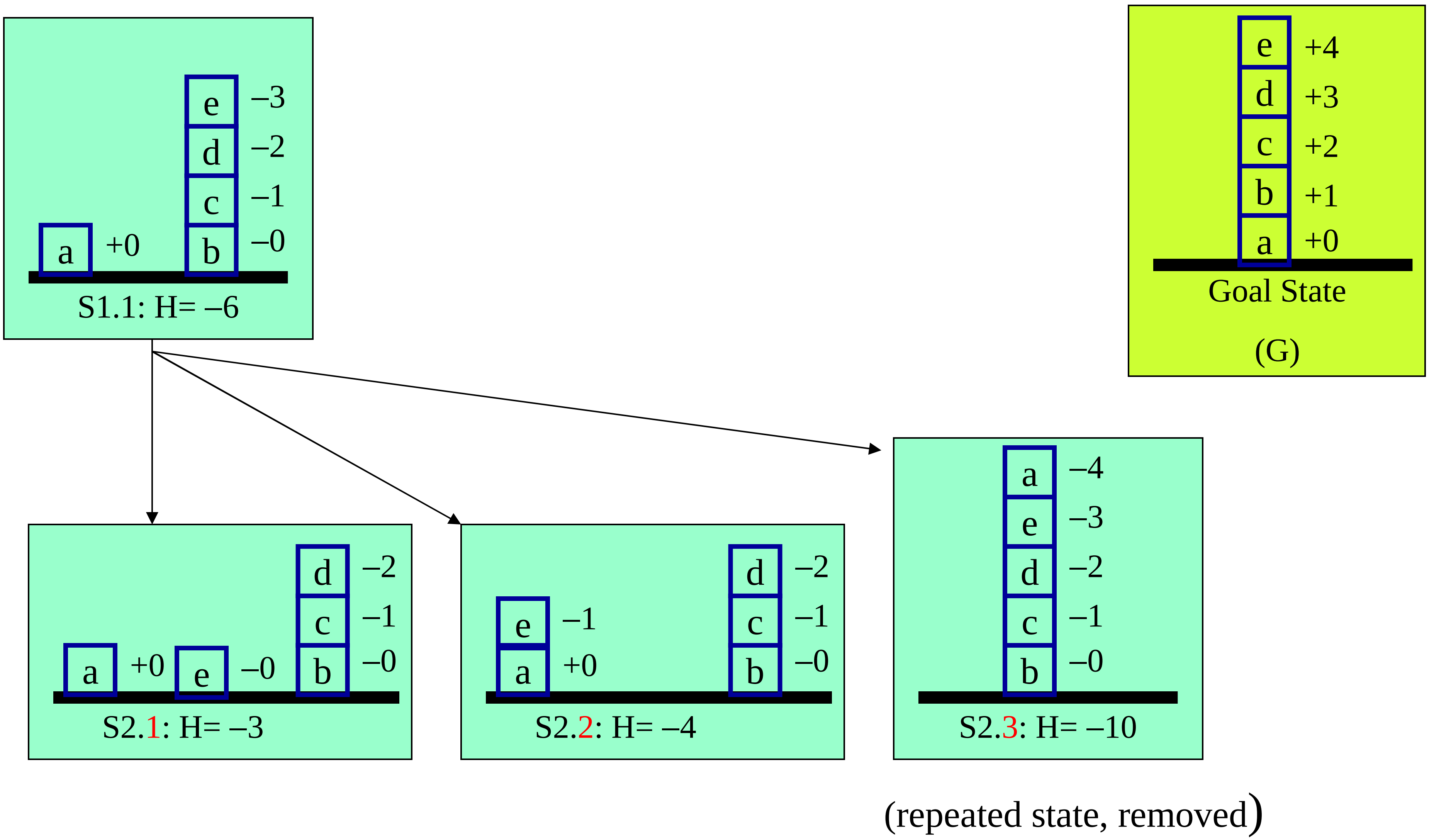
- Global heuristic, referred to as H2:
  - For every block that has the correct support structure (i.e., the complete structure underneath it is exactly as it should be)
    - ✓ +1 point for every block in the support structure
  - For every block that has the incorrect support structure:
    - ✓ -1 point for every block in the support structure


a
e
d
c
b
Start State
(S)


e
d
c
b
a
Goal State
(G)

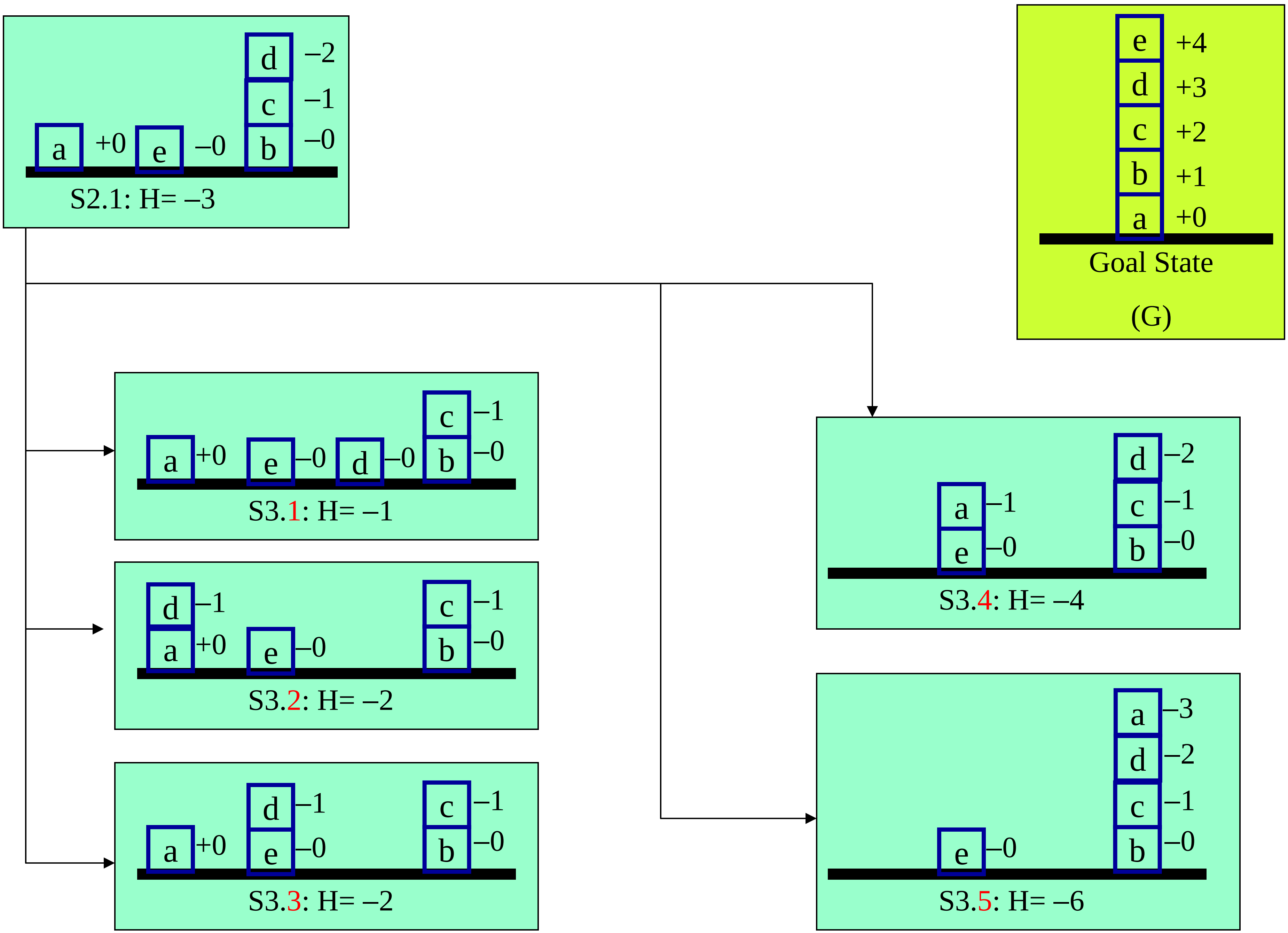


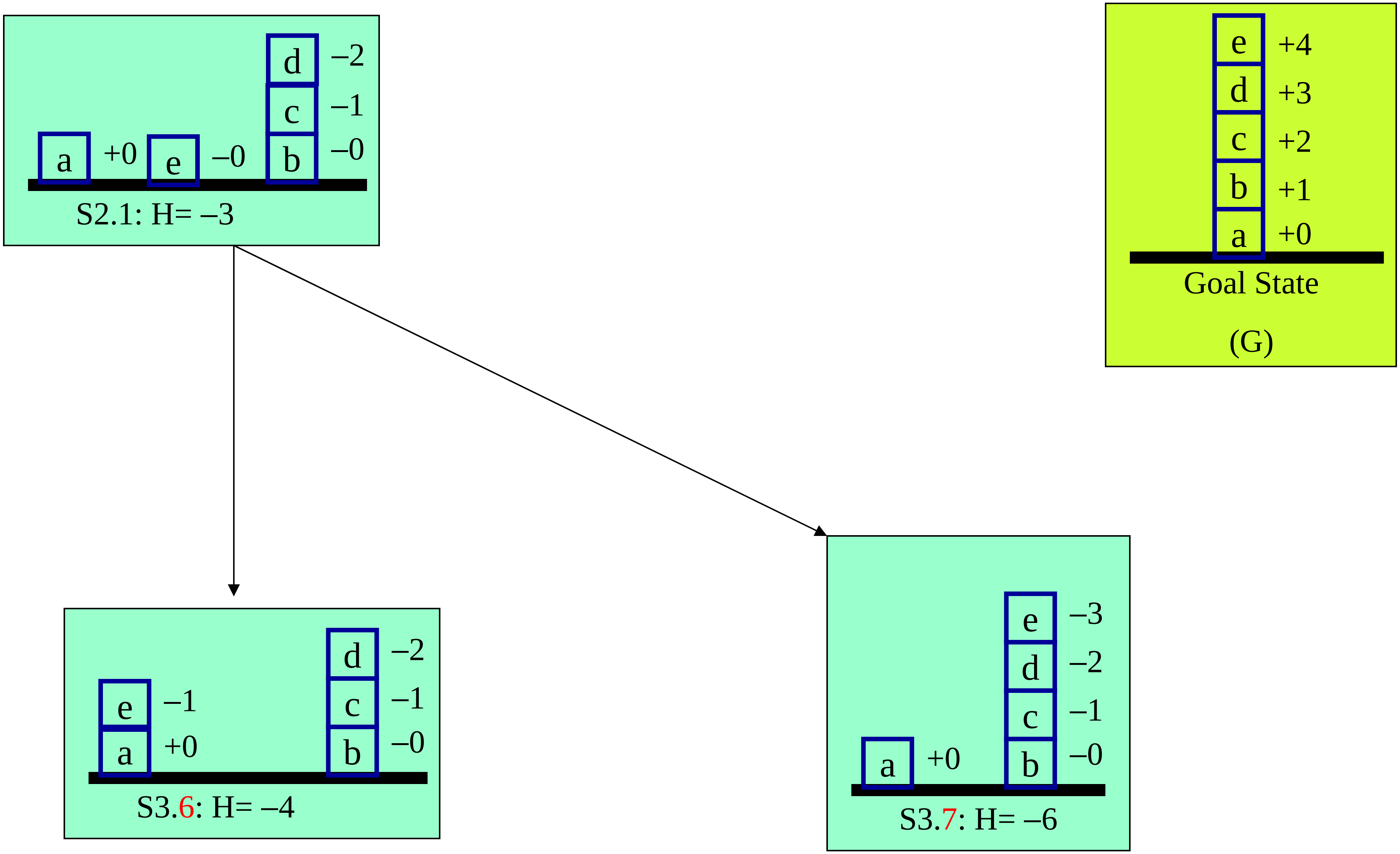
S1.1 is better than S0
→ S0 become current state



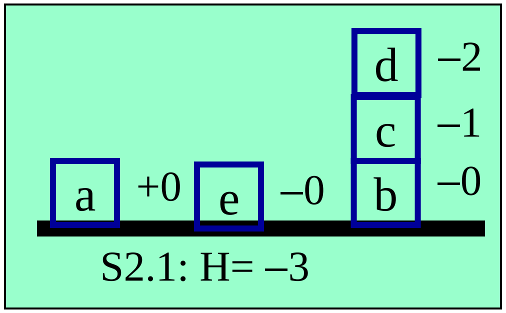
S2.1 is the best among the three states

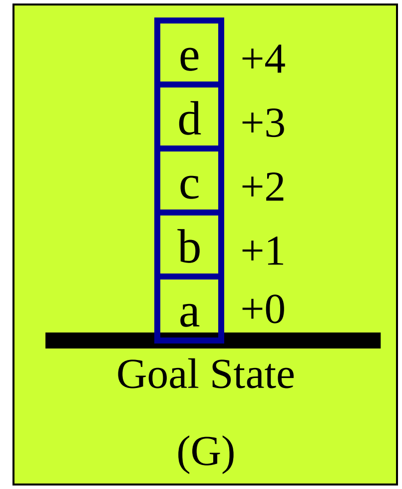
→ S2.1 becomes the current state

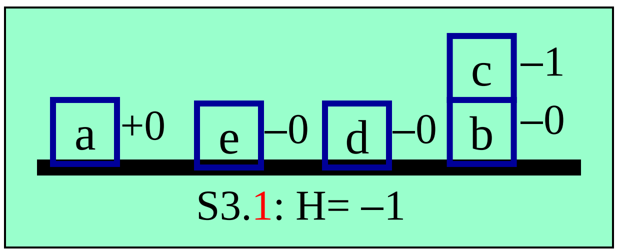

d –2
c –1
a +0
e –0
b –0
S2.1: H= –3
e +4
d +3
c +2
b +1
a +0
Goal State
(G)
c –1
a +0
e –0
d –0
b –0
S3.1: H= –1
d –2
a –1
c –1
e –0
b –0
S3.4: H= –4
d –1
c –1
a +0
e –0
b –0
S3.2: H= –2
a –3
d –2
c –1
e –0
b –0
S3.5: H= –6
d –1
c –1
a +0
e –0
b –0
S3.3: H= –2



(repeated state, removed)

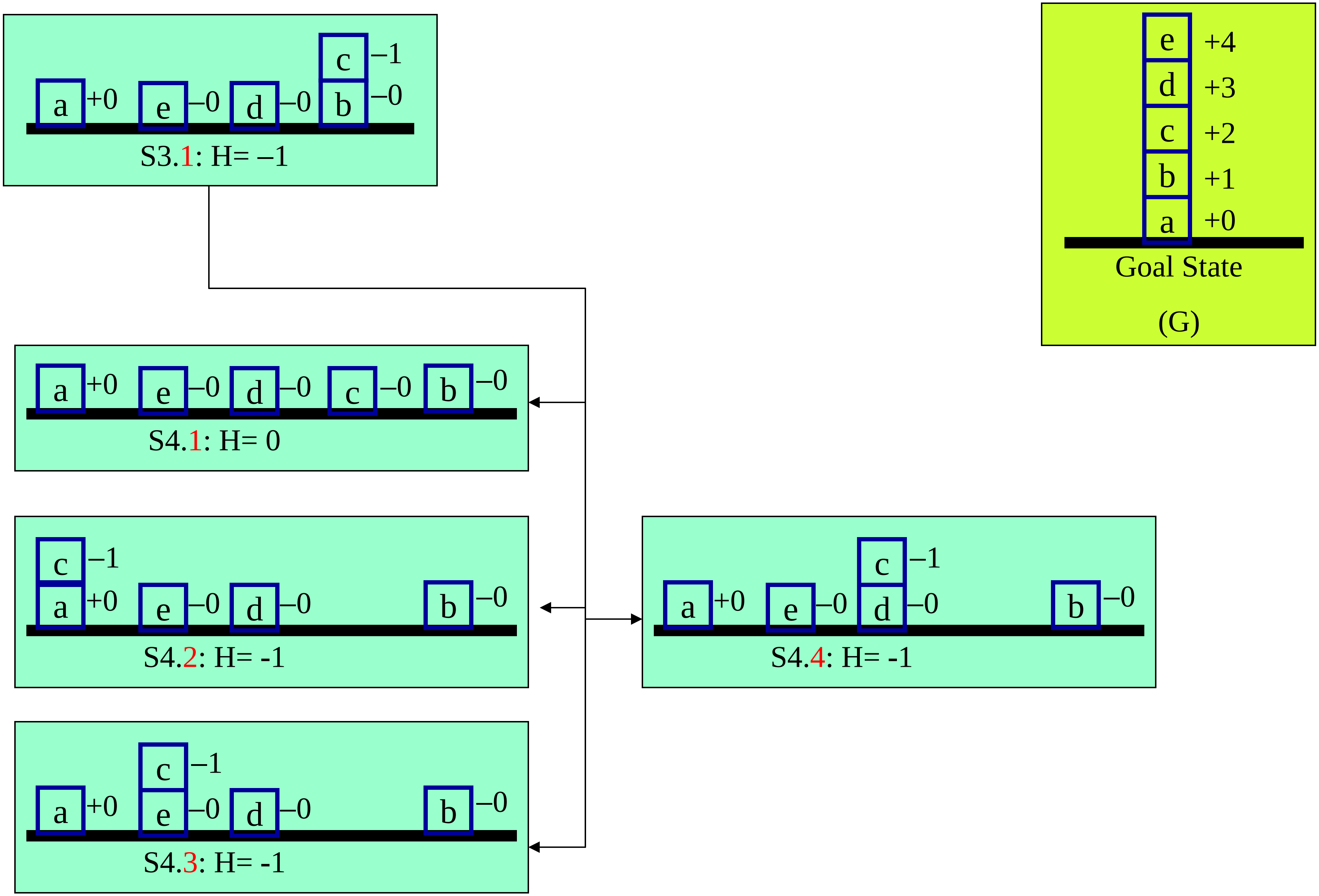


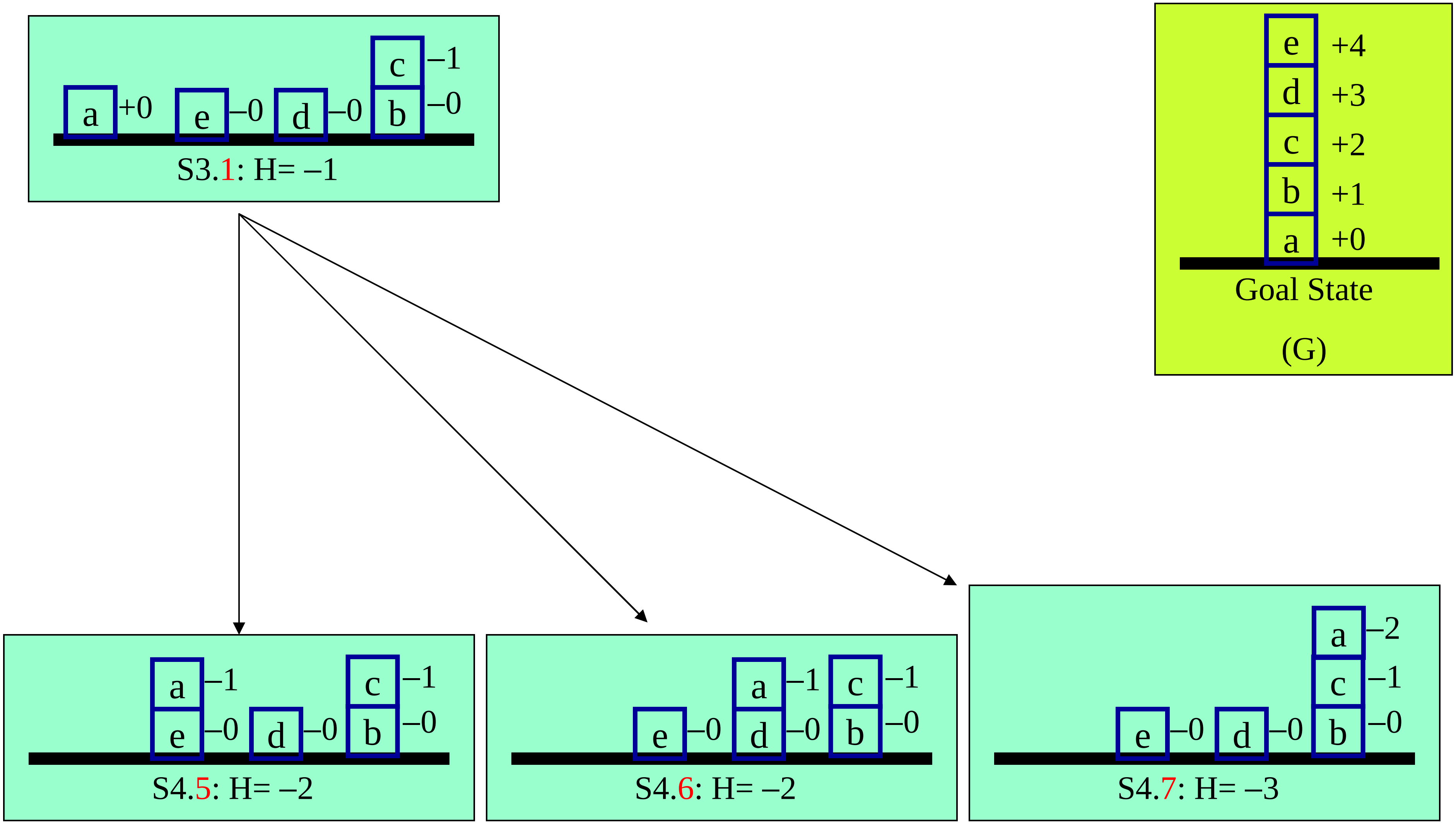


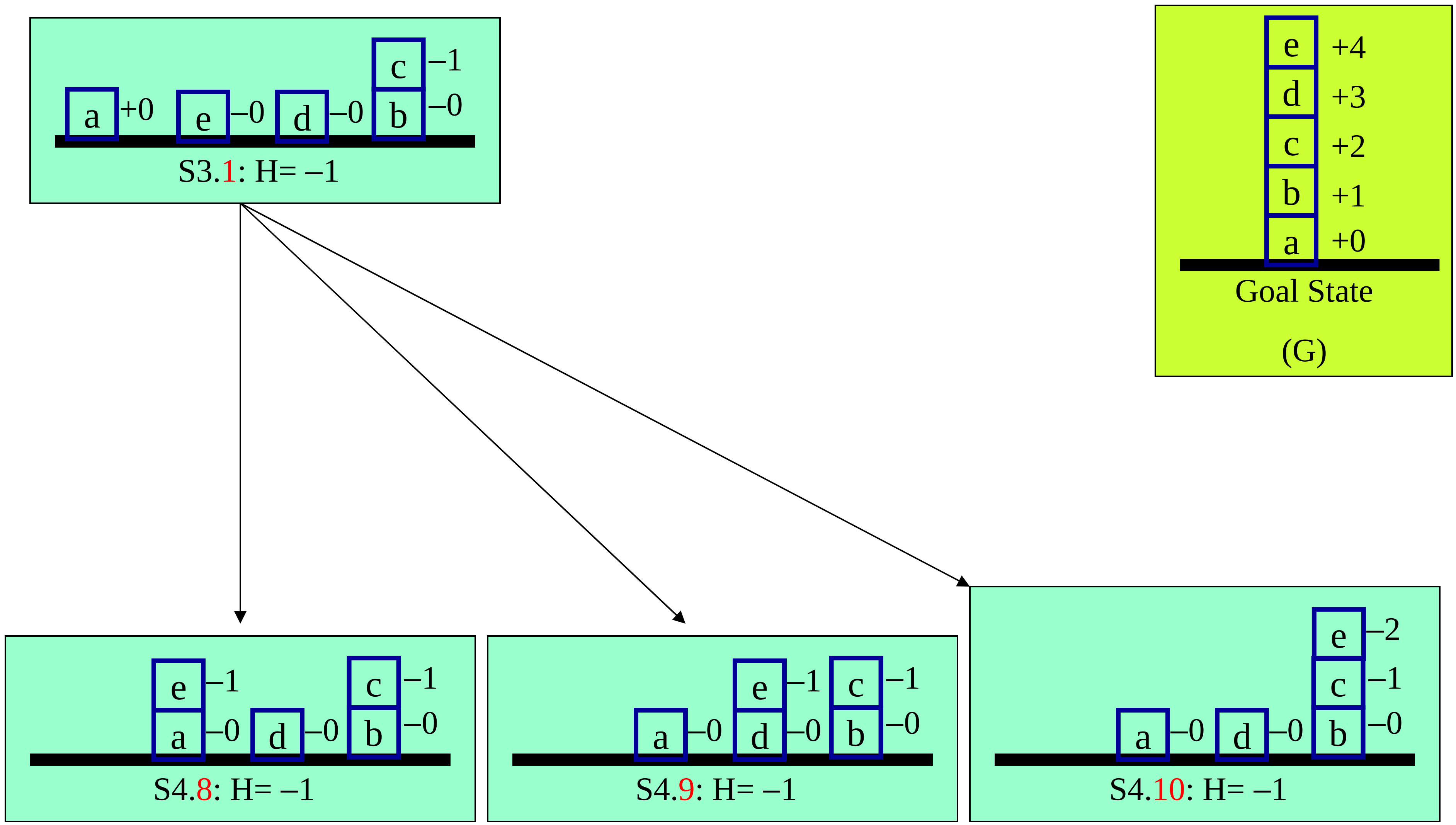


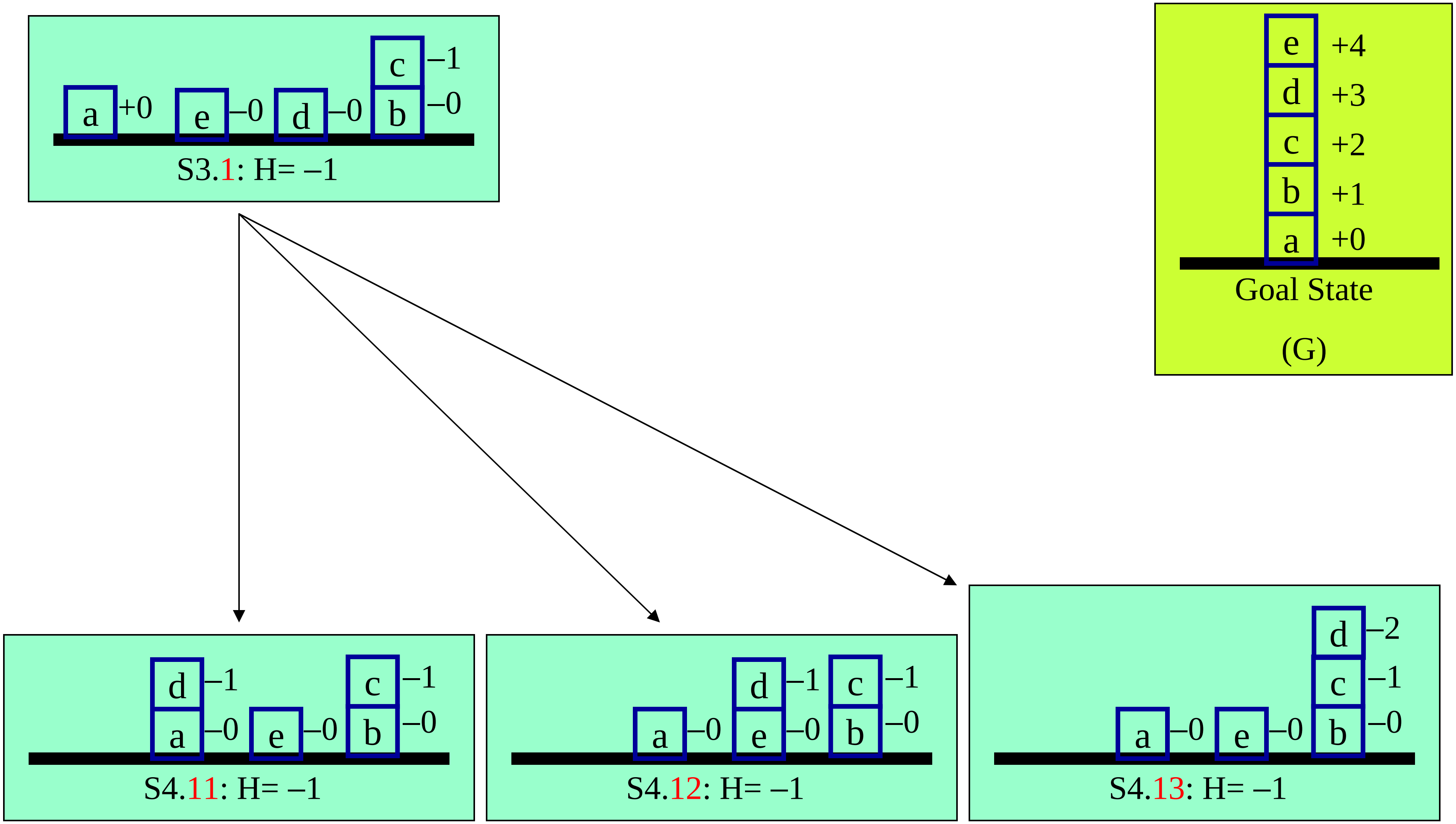
S3.1 is the best among its siblings

$\Rightarrow$ S3.1 becomes the current state

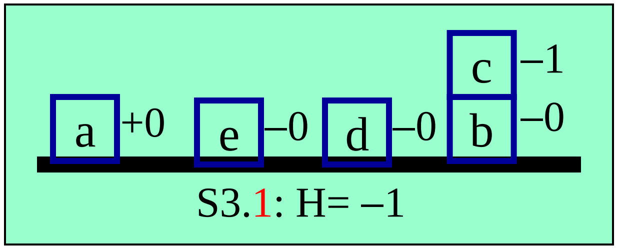

c −1
a +0 e −0 d −0 b −0
S3.1: H= −1
e +4
d +3
c +2
b +1
a +0
Goal State
(G)
a +0 e −0 d −0 c −0 b −0
S4.1: H= 0
c −1
a +0 e −0 d −0 b −0
S4.2: H= -1
c −1
a +0 e −0 d −0 b −0
S4.4: H= -1
c −1
a +0 e −0 d −0 b −0
S4.3: H= -1

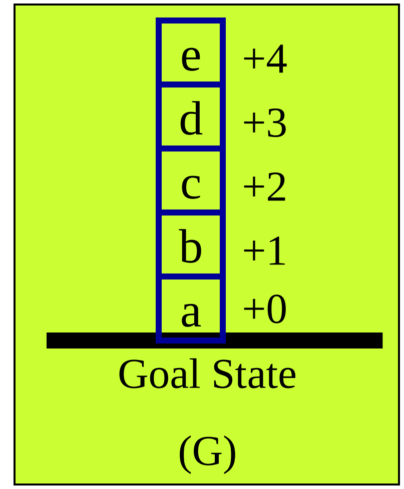

S3.1: H= –1
a +0
e –0
d –0
c –1
b –0
e +4
d +3
c +2
b +1
a +0
Goal State
(G)
S4.5: H= –2
S4.6: H= –2
S4.7: H= –3
a –2
c –1
b –0

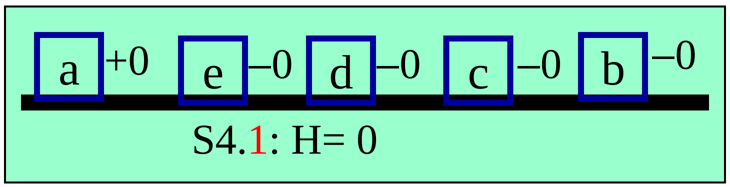

c –1
a +0 e –0 d –0 b –0
S3.1: H= –1
e +4
d +3
c +2
b +1
a +0
Goal State
(G)
e –1 c –1
a –0 d –0 b –0
S4.8: H= –1
e –1 c –1
a –0 d –0 b –0
S4.9: H= –1
e –2
c –1
a –0 d –0 b –0
S4.10: H= –1



(repeated state, removed)



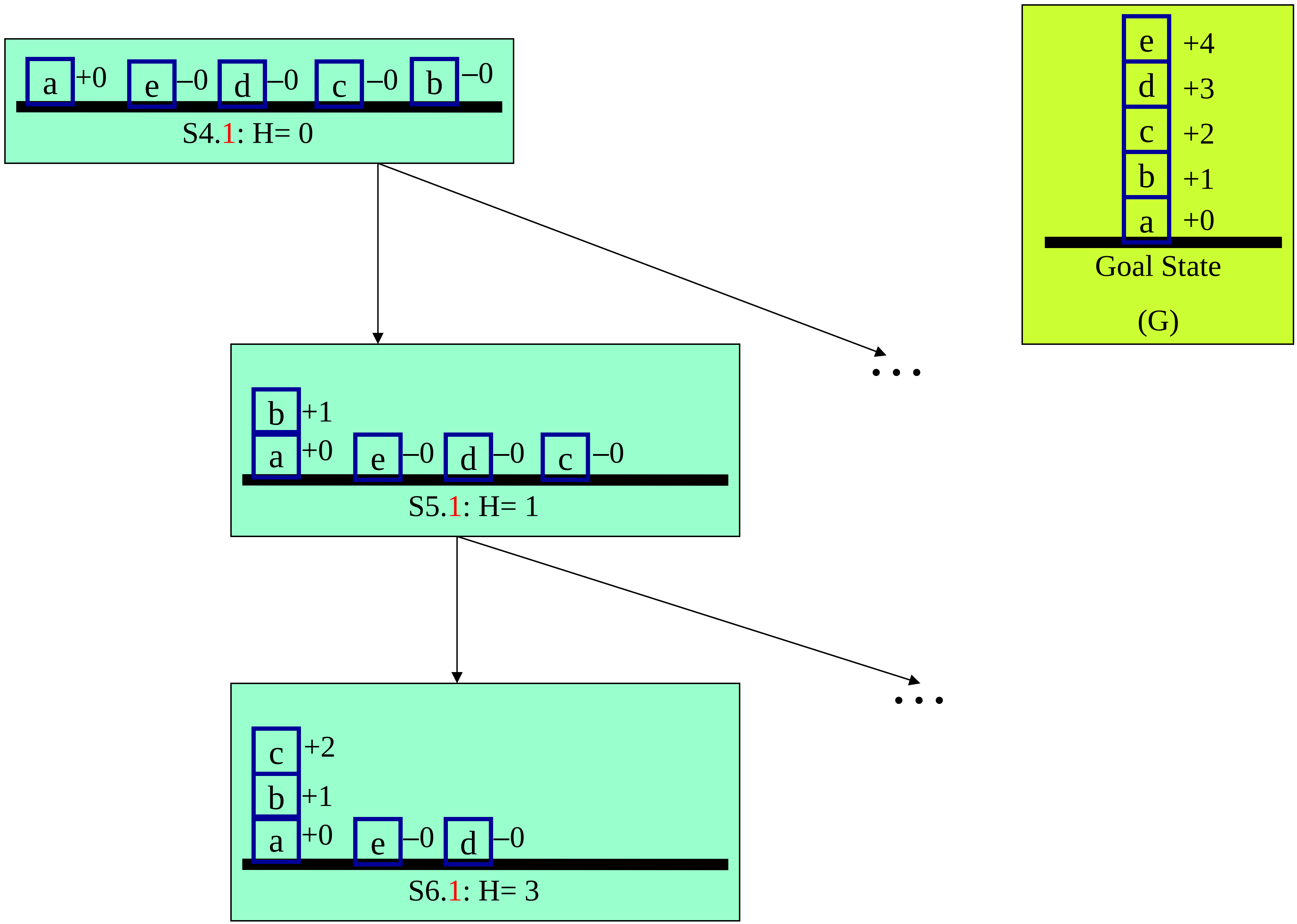


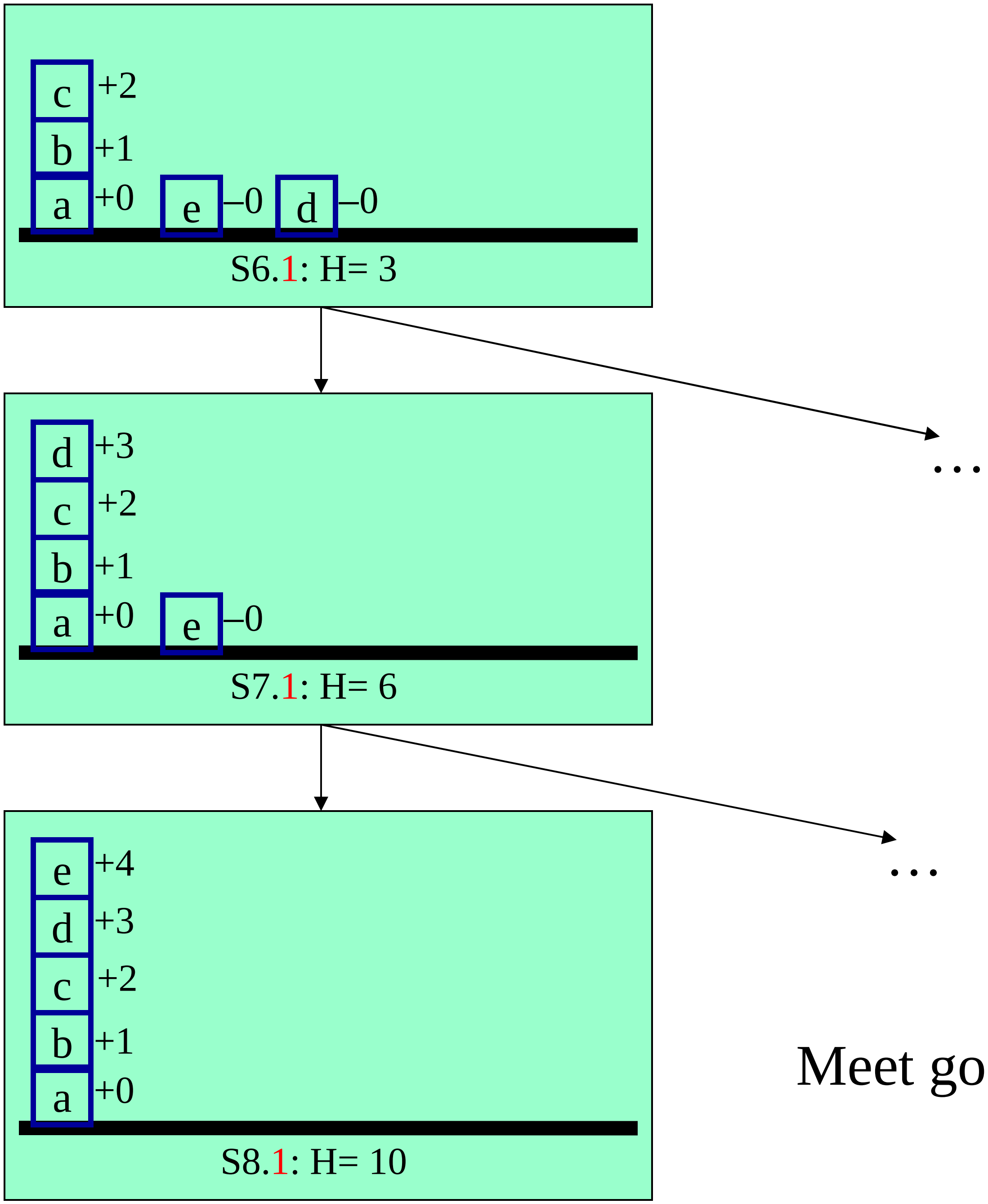


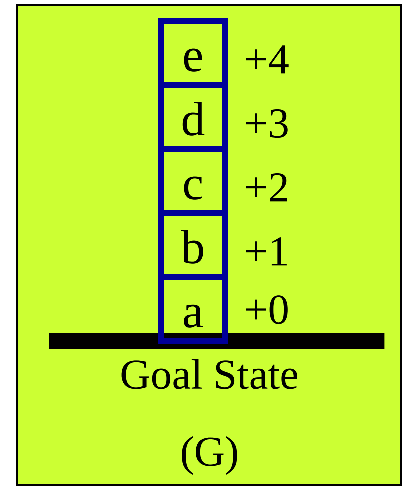
S4.1 is the best among its siblings

$\Rightarrow$S4.1 becomes the current state

Perform the similar computation as above to derive the goal state as follows.


a +0 e –0 d –0 c –0 b –0
S4.1: H= 0
b +1
a +0 e –0 d –0 c –0
S5.1: H= 1
c +2
b +1
a +0 e –0 d –0
S6.1: H= 3
e +4
d +3
c +2
b +1
a +0
Goal State
(G)





Meet goal state → Stop

N-Puzzle

- N=8:
  - Will be shown by using a demo

N-Queens

- N=5:

		Q		
Q				
			Q	Q
	Q			

Start State

Goal State
?
(Any)