GENERAL PROBLEM SOLVING WITH SEARCH ALGORITHMS

8-PUZZLE PROBLEM SOLVING

Input:

1	2	3
	4	5
7	8	6

Goal:

1	2	3
4	5	6
7	8	

8-PUZZLE PROBLEM SOLVING

Input:

1	**2**	
4	5	3
7	8	6

2 possible moves:

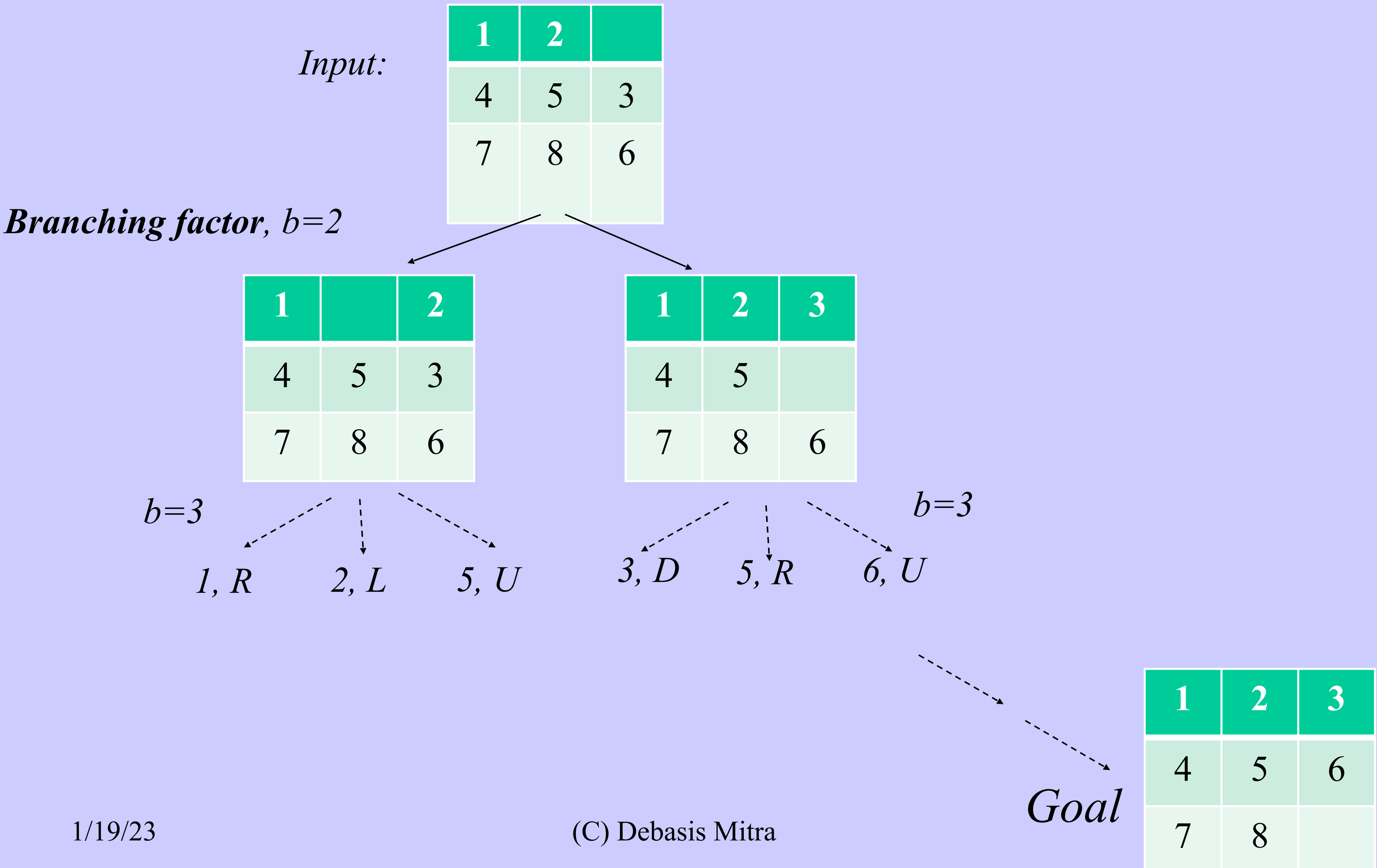
1		**2**
4	5	3
7	8	6

1	**2**	**3**
4	5	
7	8	6

1	**2**	**3**
4	5	6
7	8	

Goal

8-PUZZLE PROBLEM SOLVING



Problem Solving = Graph Search

- **Search algorithm creates a *Search Tree***
- ***Branching Factor* may increase or decrease, *for the above*: 2, 3, 4**
- **Tree Search: Same pattern may be repeated!**
    - **That may mean looping**
- **Really, Graph Search: use memory to remember visited nodes**

Problem Solving = Graph Search

- **Note, in AI it is being <<dynamically>> performed: *Generate-and-Test***
  - ***Test* in each iteration, if solution is reached**
  - **Search tree is dynamically getting *generated* during search**

  - ***Generate-test (AI-search) = Map-reduce (LISP) = Map-reduce (Hadoop-BigData)***

- ***AI search:* The Graph may NOT be *statically* available as input**

- **Some input board may not even have any solution!**

Problem Solving = Graph Search

- **Problem Solving in AI is often (always!): Graph Search**
  - **It is mostly graph search, but often we treat it as a tree search,**
  - **ignoring, repeat visit of nodes**
    because memorizing & checking for past-visited nodes is too expensive!
- **Each node is a "state":** ***State space of nodes is searched***
- ***Control types:***
  - ***Breadth First Search (BFS)***
  - ***Depth First Search (DFS)***

Sliding Puzzle Solving with BFS/DFS

Input or Problem instance

1	**2**	
4	5	3
7	8	6

Goal

1	**2**	**3**
4	5	6
7	8	

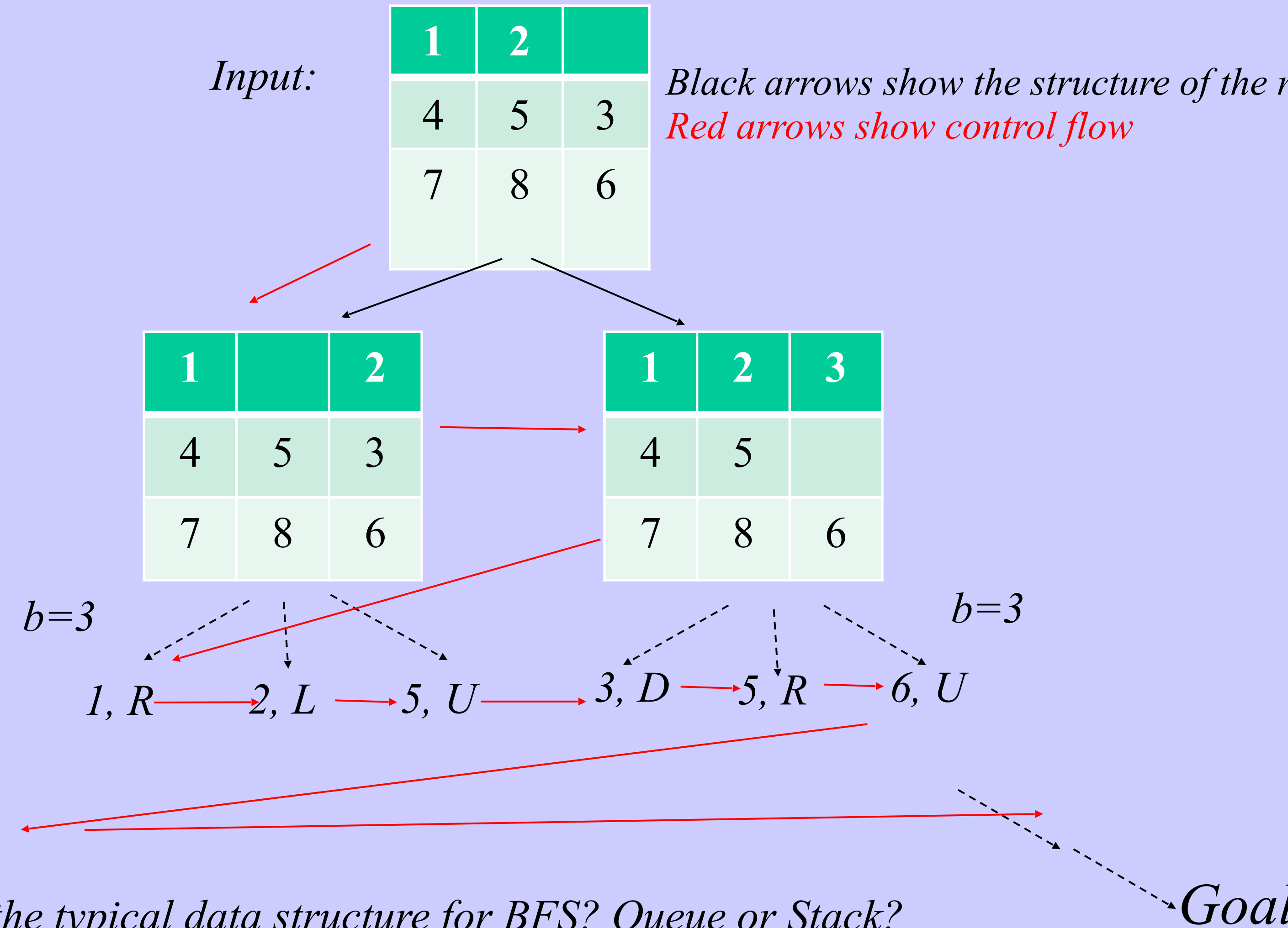
- ***Think of input data structure.***
- **Input Size *n*=**
  - ***8-Puzzle: 3x3 slots, n= 9***
  - ***15-Puzzle: 4x4 slots, n=16***
  - ***…***
- ***Problem Complexity= Total number of feasible steps over a search tree***
  - ***What are the steps: Left/Right/Up/Down***
  - ***Complexity as O(f(n))***
  - ***#Steps: $O(b^n)$, for average branching factor b:***
    - ***Branching means, how many options available at each step***

AI Problem Solving

- ***NP-hard problems: likely to be O(k^n), for some k>1, exponential***
    - ***Clever, efficient algorithms exist, but worst case remains exponential***
    - ***AI is often about efficiently solving with clever algorithm!***

BFS: 8-PUZZLE PROBLEM SOLVING

Black arrows show the structure of the recursion tree
Red arrows show control flow



What is the typical data structure for BFS? Queue or Stack?

Breadth First Search

Algorithm BFS(s): s start node
0. Initialize a Q with s;

1.v = Pop(Q); *// BFS is Queue based algorithm*
2.If v is "goal" return success;
3.mark node v as visited; *// done!*
// absent in "tree" search mode

4. operate on v; *// e.g., evaluate if it is the goal node, if so, return*

5. for each node w accessible from node v do

6. if w is not marked as visited then *// with generate-and-test this may not be time consuming*

7. Push w at the back of Q;
end for;
End algorithm.

Code it!

DFS: 8-PUZZLE PROBLEM SOLVING

Input:

1	**2**	
4	5	3
7	8	6

Red and green arrows are control flow

1		**2**
4	5	3
7	8	6

1	**2**	**3**
4	5	
7	8	6

b=3

1, R *2, L* *5, U*

b=3

3, D *5, R* *6, U*

Until leaf node, no more children

Goal

Depth First Search

Algorithm DFS(v) *// recursive*

1.mark node v as visited; *// done!*

// absent in "tree" search mode

2.If v is "goal" return success;

3.operate on v ; *// e.g., evaluate if it is the goal node, if so, return*

4.for each node w accessible from node v do

5. if w is not marked as visited then *// again, absent in "tree" search mode*

6. DFS(w); *// an iterative version of this*

//will maintain its own stack

end for;

End algorithm.

Driver algorithm

Input: Typical Graph search input is a Graph *G*: (nodes *V*, arcs *E*)

// Here we have: a board as a node *p*, and operators for modifying board correctly

// i.e, we generate the graph as we go: *"Generate and Test"*

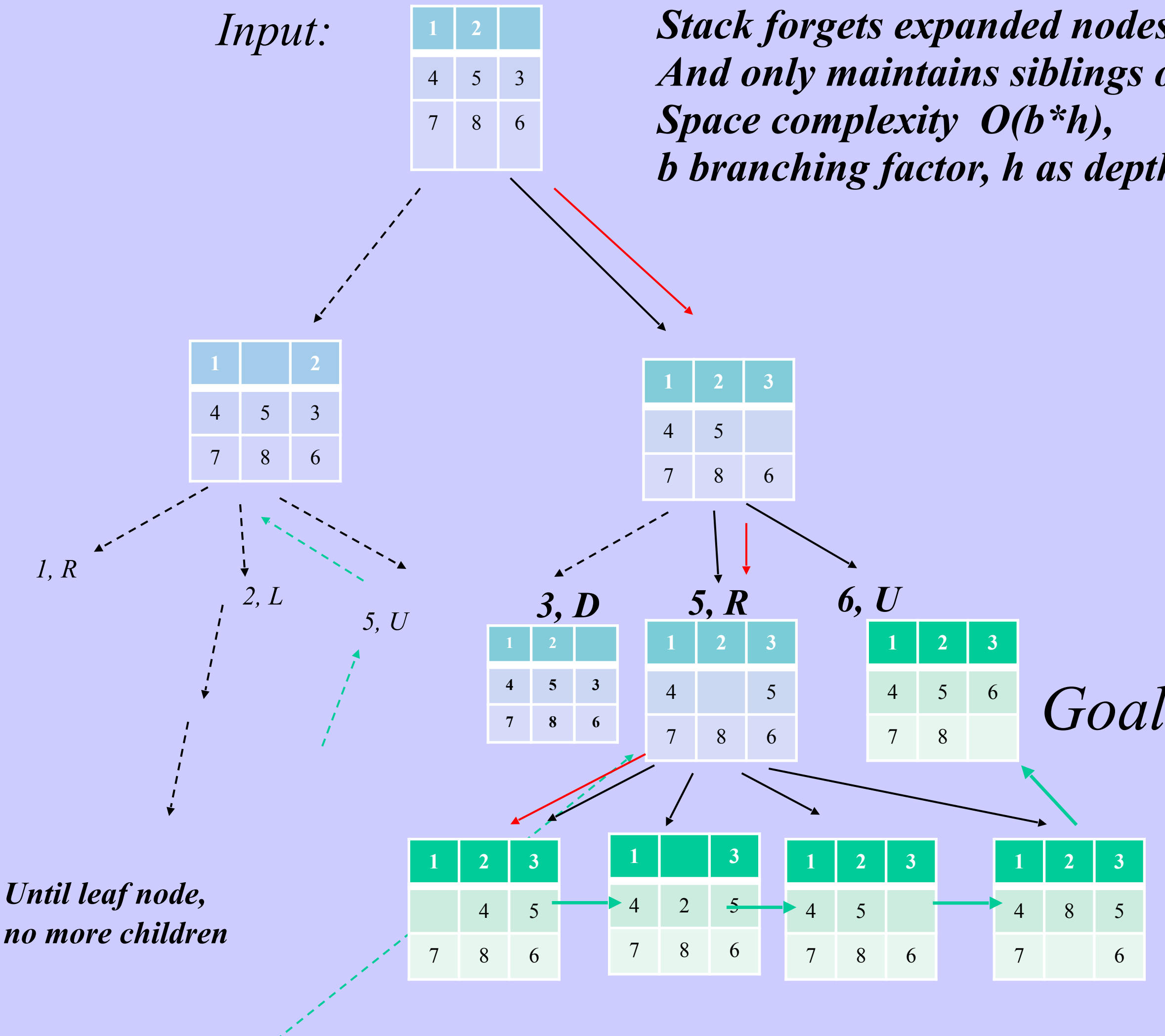
Output: Path to a goal node *g, computer may display the next node on the path*

call *DFS(p)*; *// input board p as a node in search tree G*

End.

DFS: 8-PUZZLE PROBLEM SOLVING

Stack forgets expanded nodes (transparent),
And only maintains siblings of current path:
Space complexity O(b*h),
b branching factor, h as depth of search



Until leaf node,
no more children

BFS vs DFS

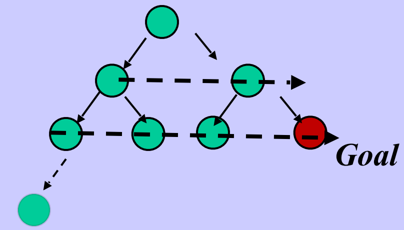
- ***BFS:***
  - ***Memory intensive: ALL children are to be stored from all levels*** ☹
    - ***At least all nodes of the last level are to be stored, and that grows fast too!***
    - ***Also, you cannot do the above if the path from start node to goal is needed!***
  - ***If goal exists, BFS is <u>guaranteed</u> to find it: complete algorithm***
    - ***(systematically finds it, level by level)*** ☺
  - ***If goal is <u>nearby</u> (at a shallow level), BFS quickly finds it*** ☺
  - ***Memory intensive, all nodes in memory*** ☹
  - ***<u>Queue</u> for implementation***

- ***DFS:***
  - ***Infinite search may happen, in the worst case*** ☹
    - ***May get stuck on an infinite depth in a branch,***
    - ***Even if the goal may be at a shallow level on a different branch***
  - ***Linear memory growth, depth-wise*** ☺ ***WHY?***
    - ***but, go to the point number 1 above → memory may explode for large depth!***
  - ***<u>Stack</u> for implementation (equivalent to recursive implementation)***

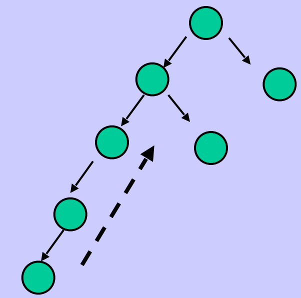
BFS vs DFS

- ***Time complexity: Worst case for <u>both</u>, all nodes searched: O(b^d)***
  b branching factor,   d depth of the goal

- ***Memory:***
  - ***BFS O(b^d) remember all nodes***

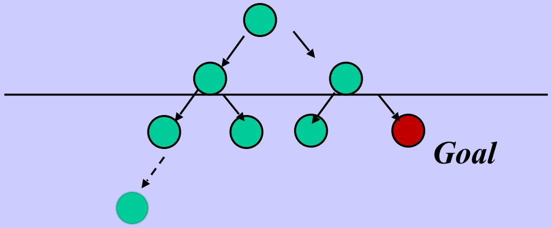


  - ***DFS O(bd), only one set of children at each level, up to goal depth***
    - ***But the depth may be very large, up to infinity*** ☹
    - ***DFS <forgets> previously explored branches*** ☺



Depth Limited Search (DLS):
To avoid infinite search of DFS

- **Stop DFS at a *fixed* depth l, no mater what**
- **Goal, may NOT be found: *Incomplete Algorithm***
  - **If goal depth $d > l$**
- **Why DLS? To avoid getting stuck at infinite (read: large) depth**
- **Time: $O(b^l)$, Memory: $O(bl)$**



Iterative Deepening Search (IDS)

- ***Repeatedly*** **stop DFS at a fixed depth** l **(i.e., DLS), AND**
  - **If goal is not found,**
  - **then RESTART from start node again, with** $l = l+1$
- **C*omplete Algorithm***
  - **Goal <<will>> be found when** $l == d$**, goal depth**
- **Isn't repetition very expensive?**
- **Time:** $O(b^1 + b^2 + b^3 + .. b^d) = O(b^{d+1})$**, as opposed to** $O(b^d)$**,**
  - **For b=10, d=5, this is 111,000 to 123,450 increase, 11%**
- **Memory:**
  - **IDS vs DFS: same O(bd)**
  - **IDS vs BFS: O(bd) vs** $O(b^d)$

Informed Search (Heuristics Guided Search)

Informed Search (Heuristics Guided Search, or "AI search")

- **Available is a "heuristic" function *f(n)* node *n*, to chose a best Child node from alternative nodes**

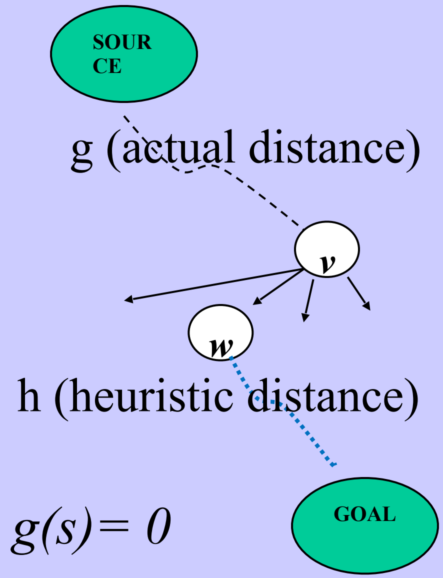
- **Example: Guessed distance from goal**
  - ***You can see the Eiffel tower (goal), move towards it!***

- **Best_child = Child *w* with minimum *f(w)***
  - **We will use *h(w)* for node *w* as guessed distance to goal**

- **AI-searches are optimization problems**

- **Both BFS and DFS may be *guided*: Informed search**

Best First Search



Algorithm ucfs(v)

1.Enqueue start node s on a min-priority queue Q; *// distance g(s)= 0*

2.While Q ≠ empty do

3. v = pop(Q);

4. If v is a goal, return success;

5. Operate on v (e.g., display board);

6. For each child w of v,

7. Insert w in Q such that cost(w) is the lowest;

// cost(w) is path cost from the child node w to a goal node,

// replacing cost(w) with g(w), path-cost of w from source, makes it Djikstra's algorithm

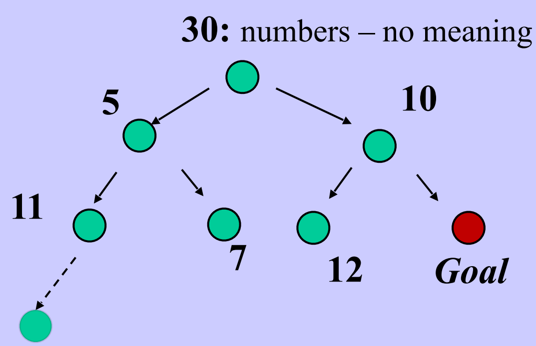
// Q is a priority-queue to keep lowest cost-node in front

Uniform Cost Search:

Breadth First with weights rather than levels

Algorithm ucs(v)
1.Enqueue start node s on a <u>min-priority</u> <u>queue</u> Q; *// distance h(s)= 0*
2.While Q ≠ empty do
3. v = pop(Q); // lowest path-cost node
4. If v is a goal return success;
5. operate on v (e.g., display board);
6. For each child w of v, if w is not-visited before,
7. Insert w in Q such that g(w) is the lowest; // use priority-que or heap for Q

Status of Q on BFS:
(when g is not used)
Q: 30
Q: 5, 10
Q: 10, 11, 7
Q: 11, 7, 12, G
Q: 7, 12, G, x
Q: 12, G, x, …
Q: G, x, …
Success



Status of Q on ucbfs:
Q: 30
Q: 5, 10
Q: 7, 10, 11 // because it is a heap
Q: 10, 11
Q: 11, 12,.. 10→G: Success

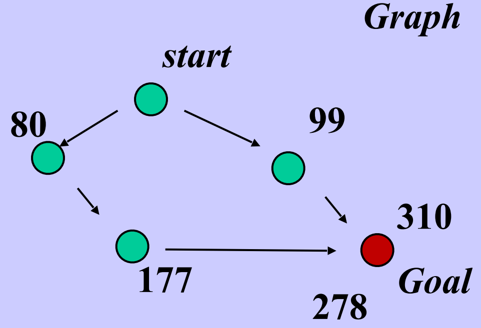
Consider mouse/robot in a maze

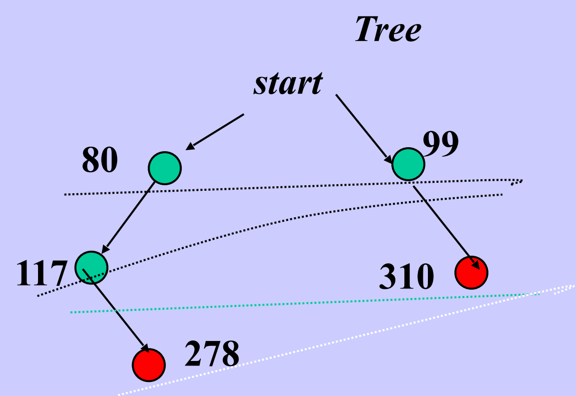
Best First Search:

Find shortest path to a goal

Similar to Djikstra's shortest path-finding algorithm (from source)

In a graph, the cost of a node may be updated via a different path



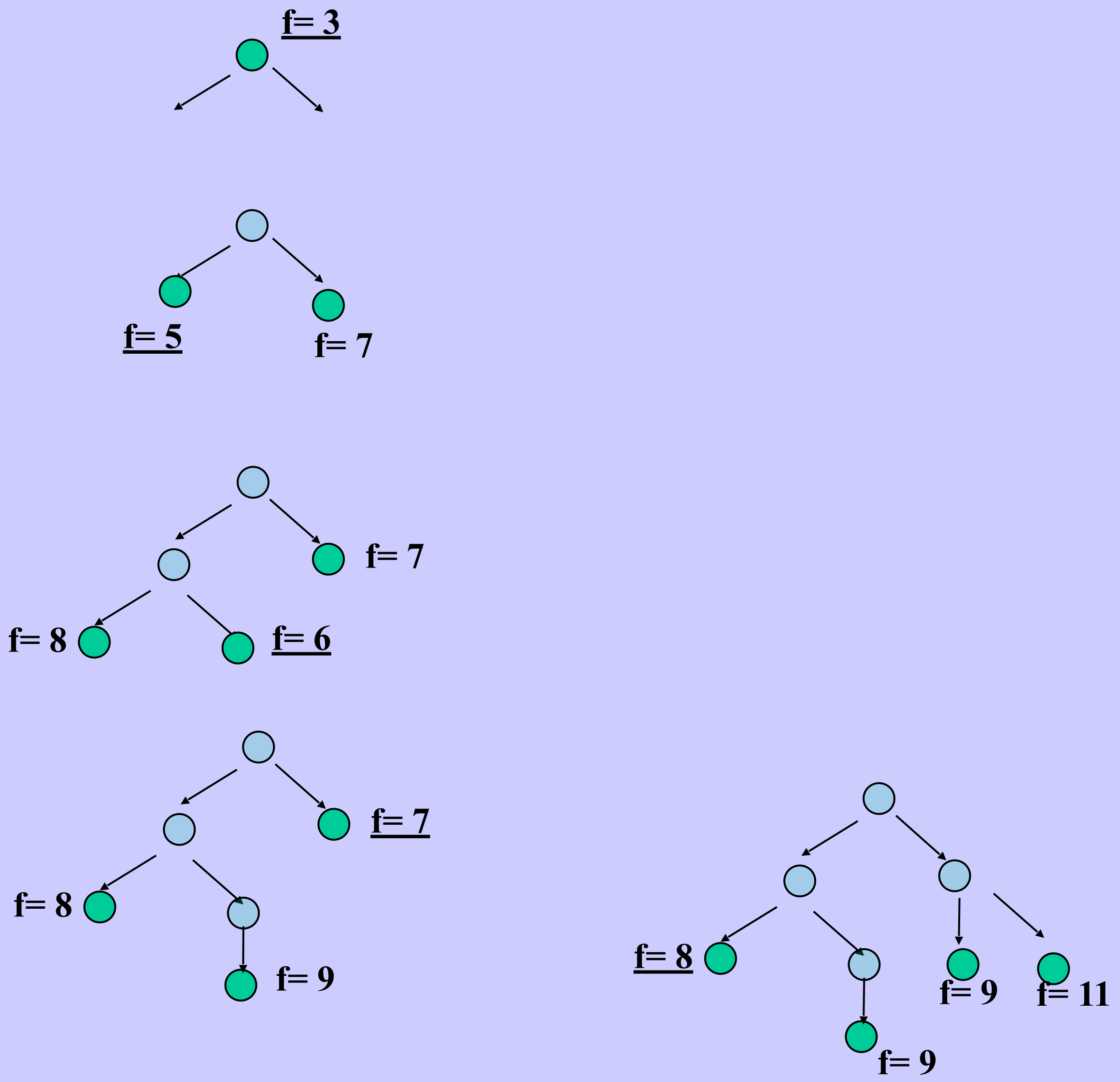


Status of Q:
Q: 80, 99
Q: 99, 177
310, Success, but continue until finished
Q: 177, 278
Q: 278
278, Success, and finished all nodes

We may stop if just finding a Goal is enough,
And shortest path is not needed,
e.g., in typical 8-puzzle problem

Animation of Best First Search

transparent nodes below are no longer in the Priority Queue



Two properties of search algorithms

- *Completeness*: does it find a goal if it exists within a finite size path?
- *Optimality*: does it find the shortest path-distance to a goal?

  - Note: not to confuse with *optimal-algorithm,* as in complexity theory:
    lowest time in Omega Ω notation
  - Our "optimal" is *optimal-search* algorithm, in AI
  - "Completeness" is somewhat similar in both the context

Greedy search: A* Search

- **“Heuristic” function** ***f(n)*** **=** ***g(n)*** **+** ***h(n)***
  g(n) = current distance from start to current node n
  h(n) = a guessed distance to goal

- ***Best-first strategy:***
  - ***Out of all frontier nodes pick up n’*** **that has minimum** ***h(n’)***
  - ***Use heap***

Mouse/robot can see a pole at the exit gate

Greedy Depth-first search: *A* Search*

- **Example:**
  - ***g(n): how far have you traveled***
  - ***h(n): how far does the Eiffel tower appear to be!***
  - ***Go toward smallest f = g+h***
    - ***least f = least path to goal***
    - ***Why least path, rather than any path?***
    - ***Because: a larger path may be infinite!***

A* Search is *Best-first with f*

- ***f*** **is *"correct"*, iff DFS is guided toward the goal correctly**
  - **"Admissible" heuristic:**

    f **is $\leq$ actual *g(goal)*, in minimize problem**

- **Say, *f(n) = 447* , then**

  there should be no path to *goal* <u>*via n*</u> costing 440

- ***f(n)*** **must be lower than TRUE distance to the goal via *n***
  - ***f(n)*** **should be an *honest guess* toward "true" value**
  - ***If f(n) over-estimates, then a minimizer algo may be wrongly guided***
  - ***Admissibility is required in order to guide correctly: for "optimality"***

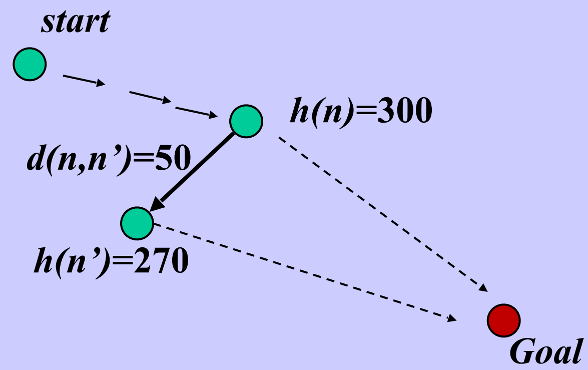
- **Why not use true distance for *f*? *Then, who needs a search* :)**
  - ***Heuristic*** **≡ You are just trying to *guess*!**

2) <u>Consistent</u> Heuristic for A* ≡ Guides only toward correct direction

- **For every pair of nodes *n, n'*,**
  such that *n'* is a child of *n* on a path from start to goal node

- **If, true distance $d(n, n') \geq h(n) - h(n')$ (*consistent*)**
  - **Means *Triangle inequality*, or**
  - **Heuristic function increases *monotonically:* $h(n')+d(n,n') \geq h(n)$**

- **Then, Guidance is perfect ≡ Never in a wrong direction ≡ *Consistent***
  - **NO backtrack necessary beyond "*depth contour*" of goal on DFS**
    - **[*backtracking wastes time*]**



$h(n) - h(n')$ =30
$d(n,n')$ =50 >30

Note: heuristic function is $f(n) = g(s \text{ to } n) + h(\text{estimated from } n \text{ to goal})$
and, $d(n, n')$ = actual distance in input between n and n'

Consequence of Consistent Heuristic for A*

- **Consistent ≡ Guidance is perfect:**
  will find a path if it exists &
  will find a shortest path

- **Search "flows" like water downhill toward the goal,**
    - **following the best path**
  - **Flows perpendicular to the contours of equal *f*-values**

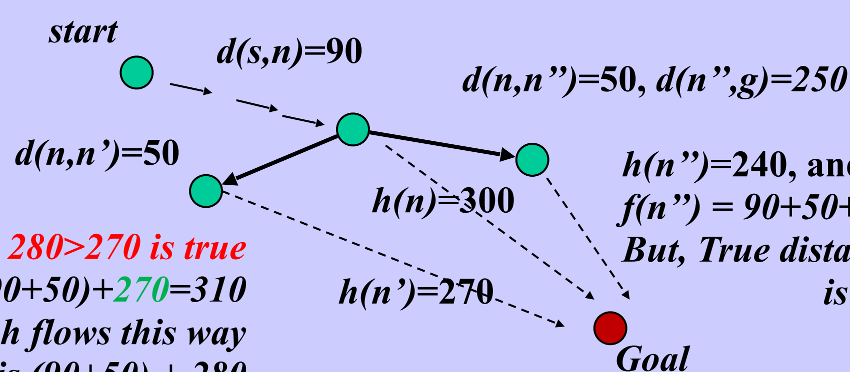
- ***Otherwise*,** // proof: book slides Ch04a sl 30

Consequence of Consistent Heuristic for A*

- If, true distance $d(n, n') \geq h(n) - h(n')$
- Heuristic function increases monotonically: $h(n)+d(n,n') \geq h(n')$

- Then, Guidance is perfect ≡ Never in a wrong direction ≡ Consistent
- NO backtrack necessary on DFS [backtracking wastes time]
- *Otherwise*,

Say, $h(n) - h(n') = 300-270=30$**, but** $h(n) - h(n'') = 300-240=60$
$d(n,n'') = 50$**, i.e. 50 not≥ 60, or heuristic is not consistent**



$h(n'')=240$, and 250>240 is true
$f(n'') = 90+50+240=380$
But, True distance to Goal is (90+50) + 250

$d(n',g)=280$, $h(n')=270$, and 280>270 is true
$f(n')=(90+50)+270=310$
Search flows this way
But, True distance to Goal on this path is (90+50) + 280
only to find later that a better path existed via n''
Backtracks => more time

*Optimality of A**

- The book has a bit of confusion on finding the goal vs. finding "a" best path to the goal

- Admissibility is *necessary*, i.e., without admissibility guidance may be wrongly directed;

  but not *sufficient*: may get into infinite loop (same f)

- Consistency (f must increase) implies admissibility as well

- For consistent heuristic DFS

  No backtracking is necessary (beyond goal f_{goal})

  - Optimality of consistency = if stuck, then no finite path to goal

- HOWEVER, generic search is NP-hard problem, there may be Exponential number of nodes within the contour of the goal

Iterative deepening and A = IDA**

- Consistent heuristic is difficult to have in real life
- Inconsistent heuristic needs backtracking on A*
  - Just as we discussed in DFS
- A misguided search *(with inconsistent heuristic)* may get stuck:
  - infinite search
  - So, what to do? Answer: IDA*

Iterative deepening and A = IDA**

- IDS + A* provides *guarantee* of finding solution: IDA*
  at the cost of repeated run of A*

- Do not let A* go to arbitrary depth

- IDA*: fixed depth A*,
  - increase depth (by *f*) incrementally and restart A*
  - Note: backtrack is allowed on A*
    but no need for that with consistent heuristic

Recursive Best-first Search

- *It is still by,* $f = g + h$
- Remember second best f of some ancestors
- If current f ever becomes > that (say, node s = *best-f-so-far*),
    - then jump back to that node s (Fig 3.27)
- Needs higher memory than DFS (more nodes in memory),
  but a compromise between DFS and BFS

*Forgetful search: SMA**

- Often memory is the main problem with A* variations

- *Memory bounded search*:
  - Forget some past explored nodes,
    - Some similarity with depth-limited search

- SMA* optimizes memory usage
- A*, but drops worst *f* node in order to expand, when memory is full
  - However, the *f* value of the forgotten node is not deleted
  - It is kept on the parent of that node
  - When current *f* crosses that limit,
    it restarts search from that parent (*Simplified MA**)

- SMA* is complete: goal is found always,
  provided start-to-goal path can fit in memory (*no magic!*)

Real life AI search algorithms

- Two more strategies in search algorithms are prevalent,
  - in addition to these search algorithms

1. *Learning*: Even from early AI days
   - Learn how to play better, success/failure as training
     - To learn *h* function, or other strategies

2. *Pattern Database*:
   - Human experts use patterns in memory!
   - IBM DeepBlue series stored and matched successful patterns
   - Google search engine evolved toward this
   - Database indexing becomes the key problem, for fast matching

Summary of informed search

- Best first search uses some f(node) to guide search
- $f(n) = g(n)$, actual distance on the path from source

  ➔ search is on circular contour
- $f(n) = g(n) + h(n)$ ➔ A* search
- Admissible heuristic, $h(n) \leq h^*(n)$, actual optimum

  ➔ A* is complete, optimal,

  but may include suboptimal path to goal,

  however, it will not stop until optimal path is found

  ➔ A* follows narrowing contour, until C* contour but no more
- Consistent, $h(n) \leq h(n_1) + d(n,n_1)$, if n is expanded before n_1

  ➔ Euclidean triangle inequality is preserved

  ➔ No suboptimal path to goal is explored

  ➔ Less node explored, most efficient situation
- Consistent means admissible, but not the other way round

Code A* search and run on the Romanian road problem, or your favorite search problem

Our Next Module: Local Search

MORE SEARCH ALGORITHMS

- <u>Local search:</u> Only achieving a *goal* is needed,
  - not the path to the goal
- Possibilities:
  - Non-determinism in state space (graph)
  - Partially-observable state space
  - On-line search (full search space is not available)

LOCAL SEARCH ALGORITHMS

- Only goal is needed, NOT the path *(8-puzzle)*
- Sometimes: No goal is provided, only how to do "better"

- Algorithm is allowed to forget past nodes

- Search on objective function space ($f(\underline{x})$, $\underline{x}$)
  - $\underline{x}$, node / state / location / coordinate, may be a vector
  - f is continuous or discrete,
    - f continuous means direction of betterment is available as *Grad(f)*

- Examples:
  - *n*-queens problem (on chess board, no queen to attack another)
    - Search space: a node – *n* queens placed on *n* columns
  - VLSI circuit layout
    - (possibly!) *n* elements connected, optimize total area
      - Or, given a fixed area, make needed connections

LOCAL SEARCH ALGORITHMS

- Hill-climbing search ➔*A Greedy Algorithm*
- Take the best move out of all possible current moves
- 8-queens problem: minimize f = # attacks: *Fig 4.3*
  - Move a queen on its column, 8x7 next moves **(8q, 7col to try for each)**

LOCAL SEARCH ALGORITHMS

- Representation may be important for efficiency
  - Queen positioned at (x, y)?

    ~ 8x8=64 possibilities: 64x8 search space
  - Alternative: Queen is linked to a column (cannot have more than one queen per column) and its position is row# y,

    7 possibilities only: 7x8 search space
  - Complexity may depend on representation

LOCAL SEARCH ALGORITHMS

- Consequence of forgetting past: may get stuck at a local minimum
  - If problem is *NP-hard*, typically exponential # local minima
- Local peak, ridge, local plateau, shoulder (max-problem):
  - No better move available, but..
  - There may exist better maxima after one/more valley
- Solution? Jump out of local minima, random move
  - 8-queens problem: random restart the algorithm
- How do you know your next optimum is better than the last one?
  - Why do you have to forget everything?
  - Just remember the last best optimum and update if necessary!
  - Still not guaranteed to find <<Global>> optimum, but $f^\wedge$ converges

- *Systematic* Random move

- If improvement *delta-e* $\equiv d >$ threshold, then make the move
  - Otherwise, ($d \leq$ threshold), generate a random number r and
  - If $r \geq \exp(d/T)$, then take the "wrong" move anyway
    - Otherwise, randomly jump to a node

- T: a constant, but from a sequence of reverse sort
  - Reduces in each iteration: *less and less random restarts*
    - E.g., 128, 64, 32, 16, 12, 8, 6, 4, 3, 2, 1

- Concept comes from metallurgy/ condensed matter physics:
  - slowly reduce temp T from high to low:
    - molecules settle to lowest energy = strongest bonding

LOCAL SEARCH ALGORITHMS: Beam search

- Keep *k* best nodes in memory, and do Hill-climbing from each
  - Same time!

- Independent *k* nodes, but they are best *k* nodes so far

- Still has a tendency to converge to a local minimum, unless
  - *k* initial samples are "good" samples in statistical sense

- Some form of evolution is taking place!
  - With *k* children producing *k* offspring
  - Why not formalize it? ➔ *Stochastic Beam Search*
  - *Stochastic Beam Search: Choose k offspring probabilistically*

LOCAL SEARCH ALGORITHMS: Genetic Algorithm: *Fig. 4.6-7*

- Further formalization of Stochastic Beam Search from Biology
  - Genetic Algorithm

- Keep independent *k* (even number) nodes in memory

- Pair up, and crossover → two new nodes
  - Needs special representation of states/nodes: see 8-q
  - Crossover position is random

- Participation in reproduction is based on objective function *f*:
  - Not all *k* nodes participate in the crossover, some are culled
  - Higher *f* means higher probability of participation

- Random mutation between iterations are allowed
  - With a very low probability (infrequently)
  - To jump out of any local optima

- *Many types of variation of GA's exist, on each of the above choices*

CONTINUOUS SPACE: LOCAL SEARCH

- <u>Numerical optimization in Math</u>
  - *a 200 year old subject*

- (*f*(<u>*x*</u>), <u>*x*</u>): <u>*x*</u> is continuous, and *f* is continuous differentiable

- Infinite search space (not a graph search), even if bound
  - (infinite states if <u>*x*</u> is real)

- SA is useful in continuous space, but not GA
  - GA needs discrete representation
  - Discretization of <u>*x*</u> is some-times used in GA for continuous space

- Local search:
  - Typically, start coordinate and the resulting path is not important

- Main help comes from the existence of derivative of *f*:
  - Second or higher order derivatives may be needed:
    - *analytic f: Cauchy-Riemann condition helps*

CONTINUOUS SPACE: LOCAL SEARCH

Optimization theory in Math

- Gradient provides direction of next move,
  - $\underline{x} \leftarrow \underline{x}$ + a *(-sign_of(Del[f($\underline{x}$)])), *f* is the objective function
  - but, how far? What value of *a*?
  - Many algorithms exist for finding *a*
  - E.g., “line search” algorithm

- Newton-Raphson:
  - 1D: $x \leftarrow x - f(x)/f'(x)$, if analytic solution exists

  - nD: derived by using *gradient(f)* = 0
    - a = Hessian, $H^{-1}(f)$,
      - // a partial-double derivative matrix, $H_{ij} = d^2 / dx_i\, dy_j$

- Conjugate gradient: faster convergence
  - better optimization for longer vision ahead
  - winds down along the valleys

- Typically, optimization needs constraints:
  - Limit the search space!
- *Linear programming*, with inequality constraints:
  - Search space is a closed polygon (lines enclosed by lines)
  - *Simplex* algorithm, *Interior-point* search algorithm
- *Quadratic Programming*
- General case: Machine learning
  - Support vector machine
  - Neural network
  - …

NON-DETERMINISTIC SEARCH SPACE

- Typically, discrete, or else ☹

- If-else on nodes,
  - condition dependent
  - You do not know what will show up on a node – plan ahead!
  - *Example: Mars rover*

- Not as bad as it seems: *And-Or tree*

- Deterministic search: ‘Or’ graph, chose a node or the other

- Non-deterministic search: All nodes may need to be looked into
  - **And-Or tree** search
  - ‘And’ on if-else node: take all options in search: *Fig 4.10*

- Non-determinism may come from failure from a move
  - Loops are possible ➔ *try and try again, until give up!*

PARTIALLY OBSERVABLE SEARCH SPACE

- *Unobservable* = completely blind = unknown search space, really! ☹
  - Not as bad as it sounds: just get the job done! *Fig. 4.9*
- *Partially observable*: think of a blind person with the stick!
  - Sensor driven search
- State space: beliefs on which nodes (a set) the agent may be in
  - *A node does not know its own exact id*
  - *Discrete space, please!*
- Action reduces possibilities or nodes, ..
  - Action may be exploratory, just to reduce uncertainty of possible nodes the agent is in
  - Converging (fast) to the goal node → you are sure you attained it!
  - See the vacuum cleaner example:
    - A few moves blindly → room is cleaned

“ON-LINE” SEARCH

- Off-line: Search space is known, at least static
  - Non-deterministic: at least possibilities are known or “static”

- On-line: Robots
  - Compute-act-sense-compute-…..
  - State space <u>develops</u> as-we-go:
    - not “thinking” algorithm, actually make the move, no way to un-commit
  - Not much different

- Strategy, heuristics,
  - but no set of nodes, may be finite/infinite