Artificial Intelligence:
Search Methods for Problem Solving

B&B - A* - wA* - Best First

A First Course in Artificial Intelligence: Chapter 5

Deepak Khemani

Department of Computer Science & Engineering

IIT Madras

The Pull and the Push

Consider the function

$$f(n) = g(n) + w \times h(n)$$

where *w* determines how much weight we give to the heuristic function.

At one end of the spectrum is $w = 0$

with the *pull* of the source

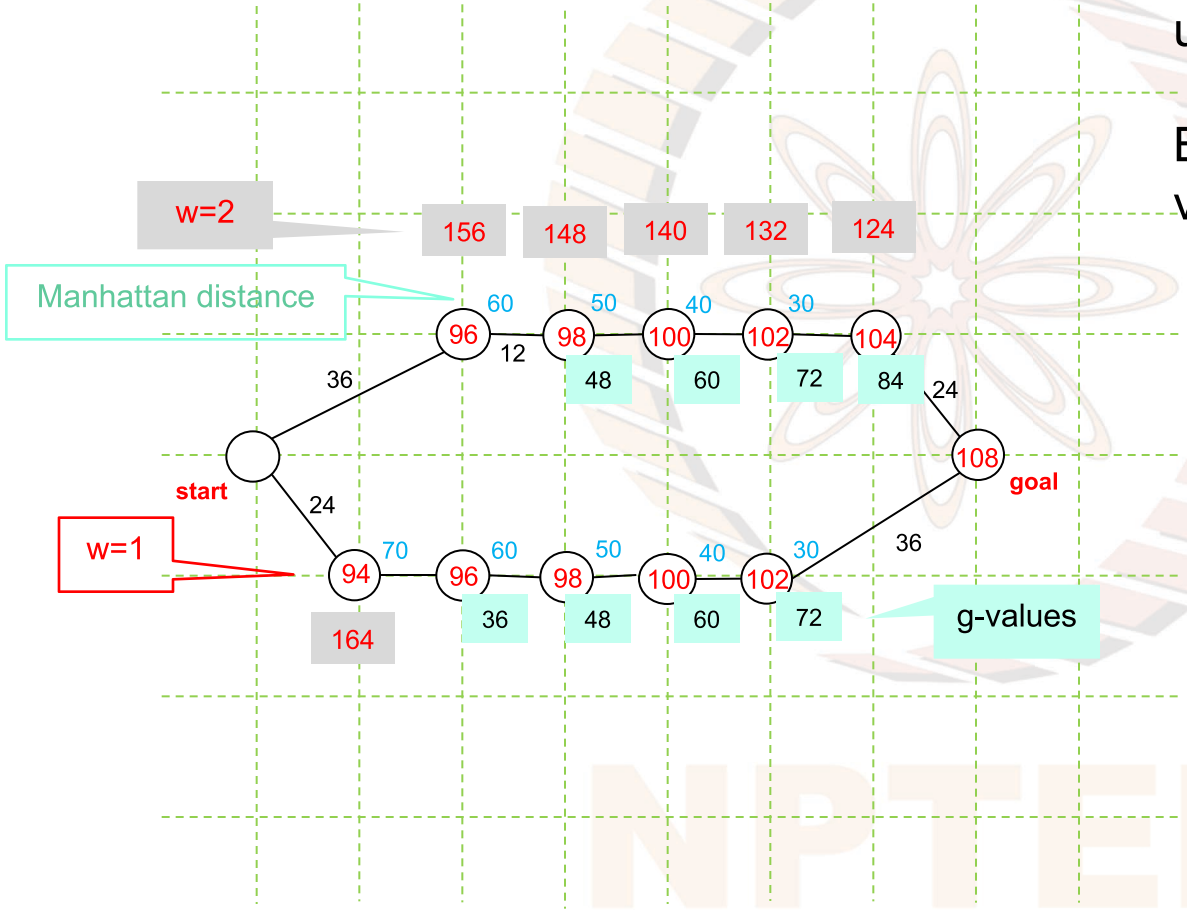
and only the shortest path in mind

At the other end of the spectrum $w \rightarrow \infty$

with the *push* towards the goal

and only speedy termination in mind

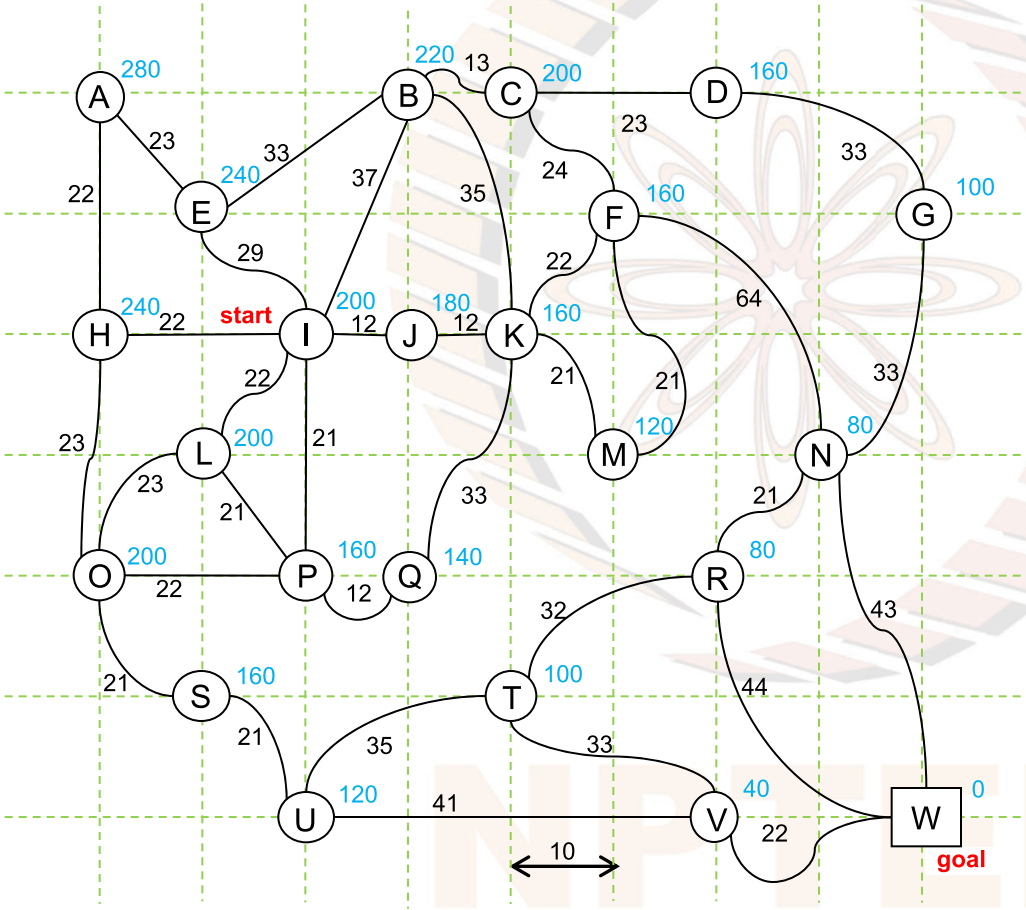
An illustrative example



Best First and wA* take the upper path

B&B and A* explore both paths visiting nodes alternately

The problem graph for wA*

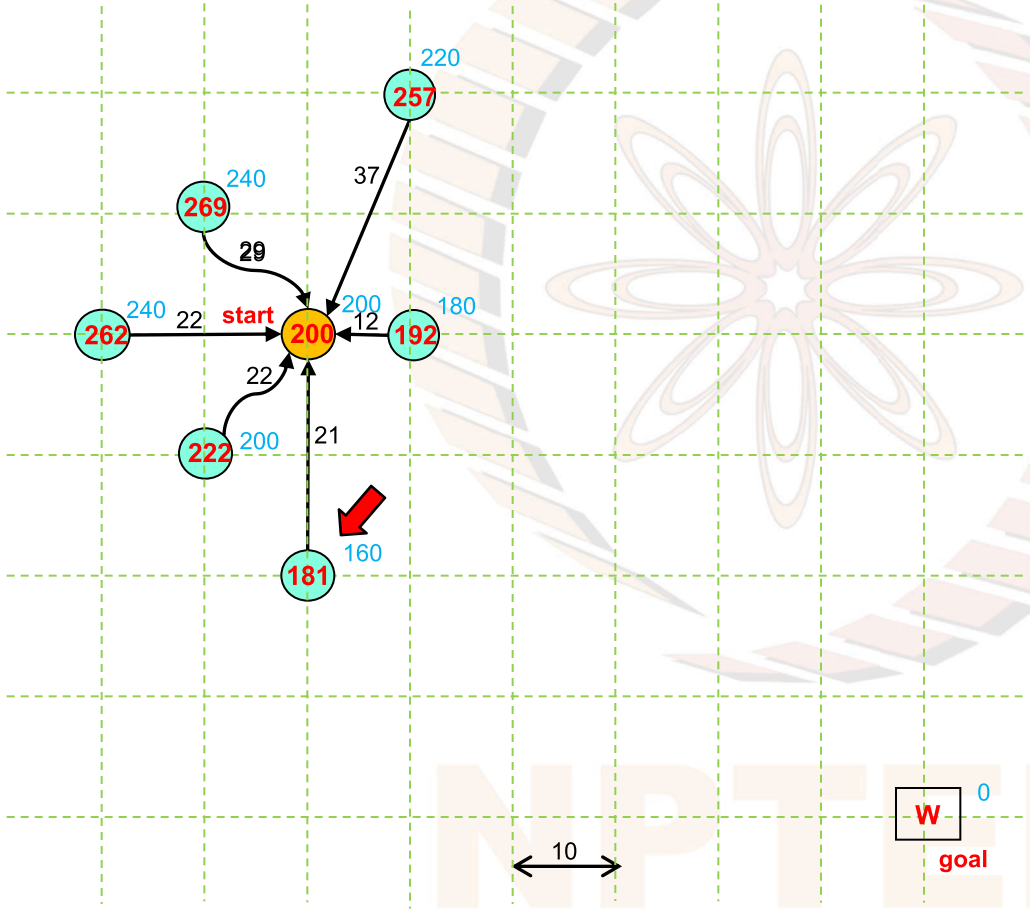


The values outside the nodes are 2 × Manhattan Distance

These are used in wA* where $f(n) = g(n) + 2 \times h(n)$

Next we trace the progress of this weighted A* algorithm

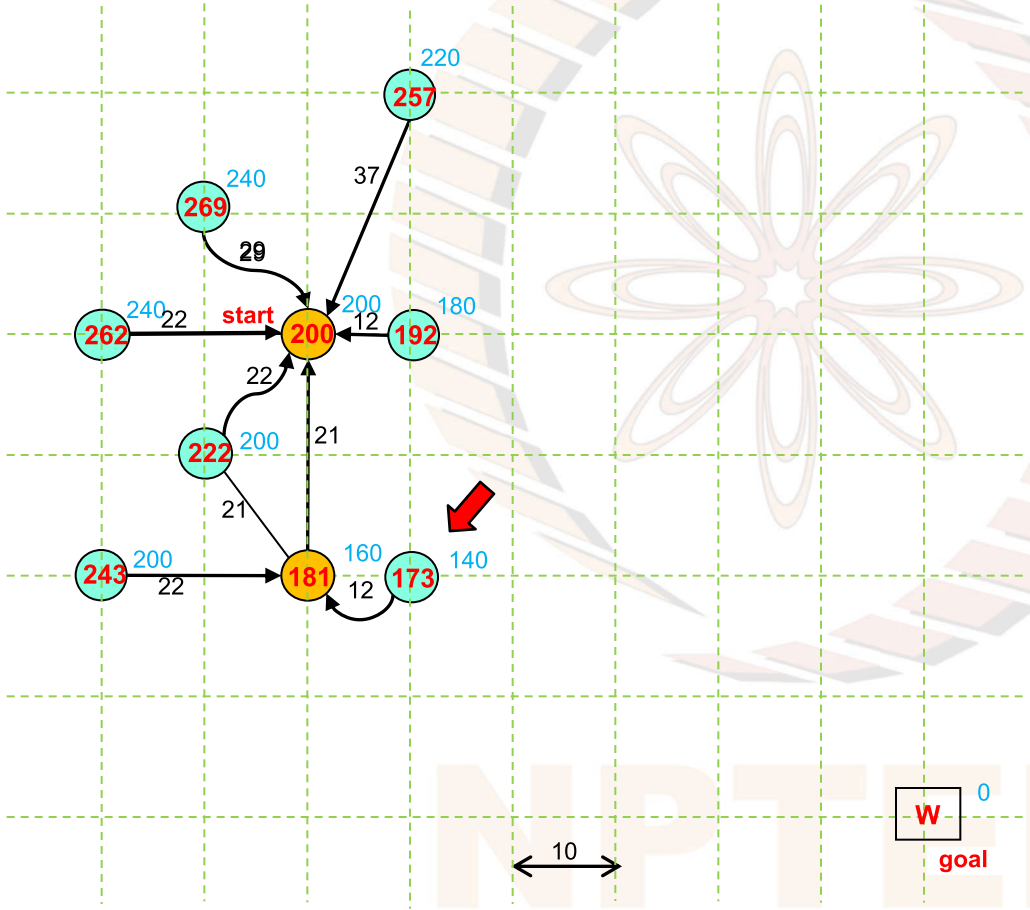
Weighted A*: f(n) = g(n) + 2 × h(n)



With weight = 2 the search algorithm gives greater importance to the heuristic value. Observe the 2 × h(n) values in cyan.

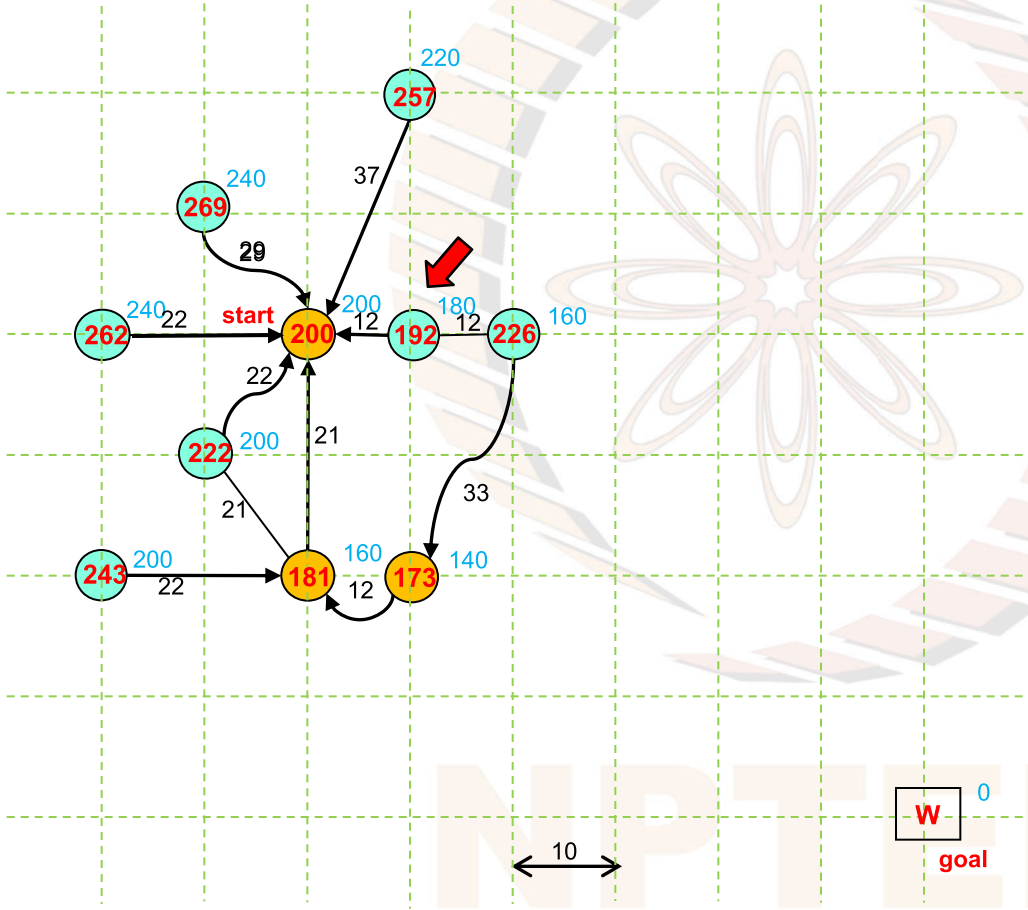
We take up the action after it has expanded the start node.

Weighted A*: f(n) = g(n) + 2 × h(n)



Initially the behavior is simililar both to A* and the Best First algorithms

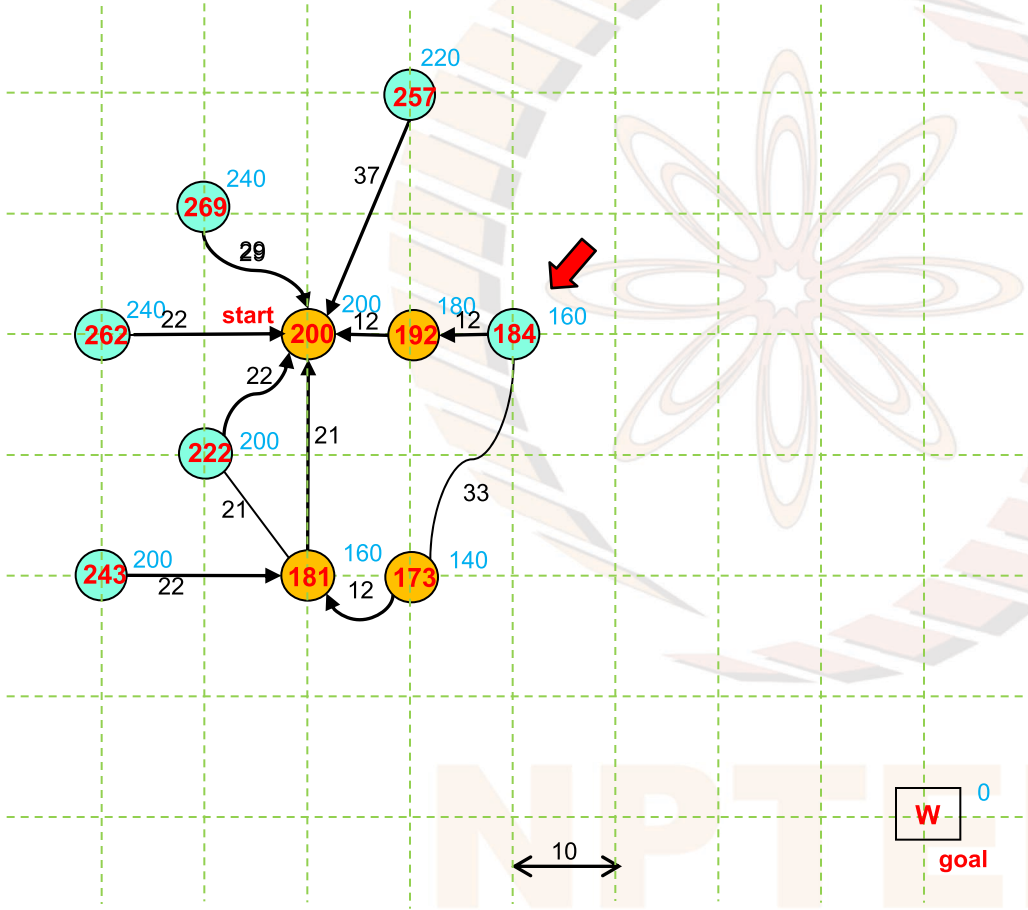
Weighted A*: f(n) = g(n) + 2 × h(n)



Unlike Best First Search the Weighted A* algorithm *does* look back to consider the g-values and prefers *this* node to its child which is closer to the goal.

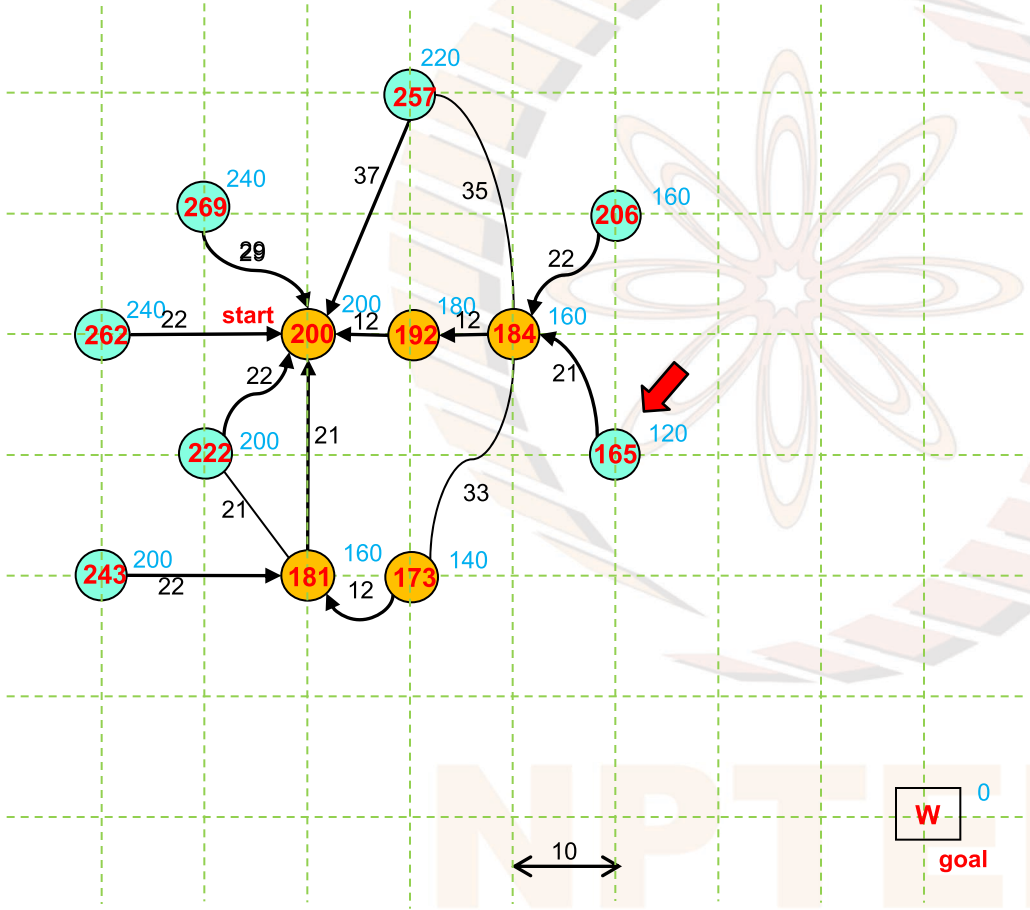
It will in fact find a better path to its child than the one already found

Weighted A*: f(n) = g(n) + 2 × h(n)



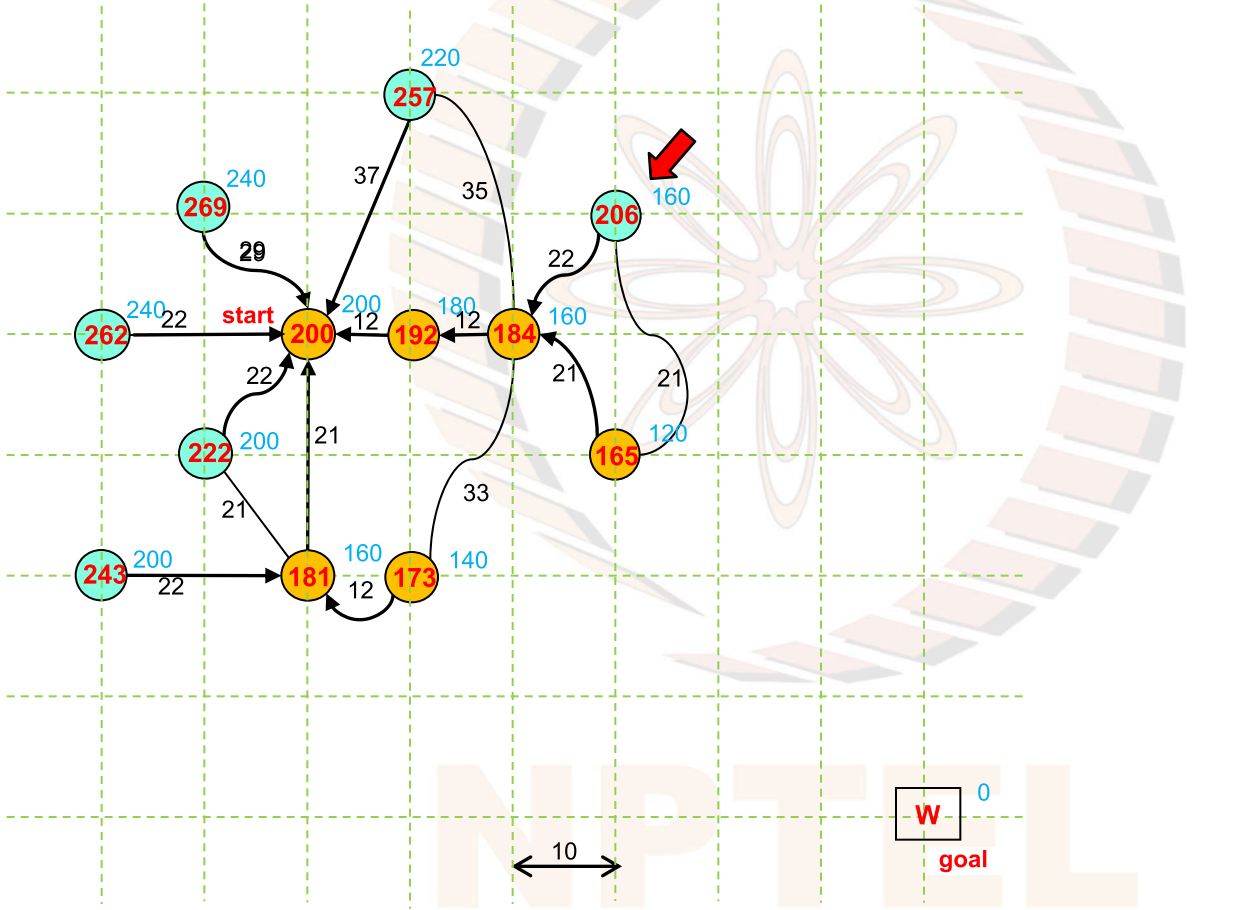
Observe that the value of *this* node has reduced from 226 to 184.

Weighted A*: f(n) = g(n) + 2 × h(n)

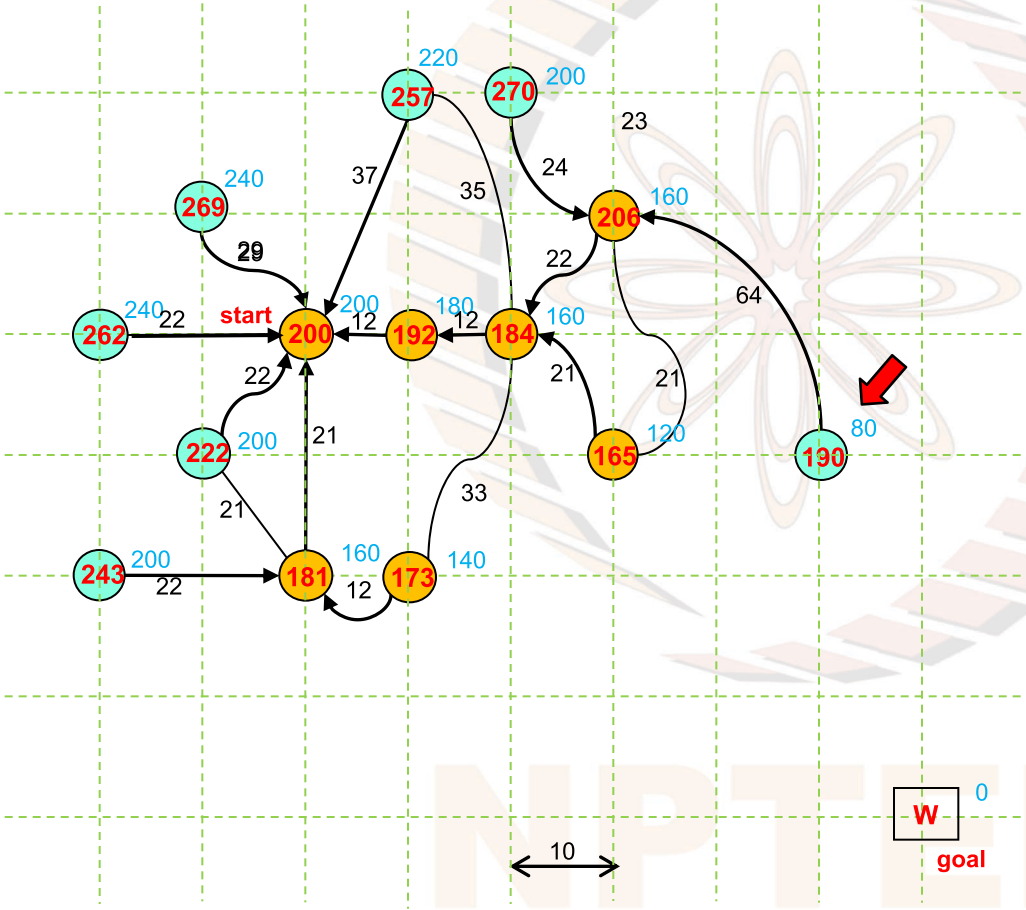


All algorithms explore this dead-end because it appears to be closest to the goal.

Weighted A*: f(n) = g(n) + 2 × h(n)

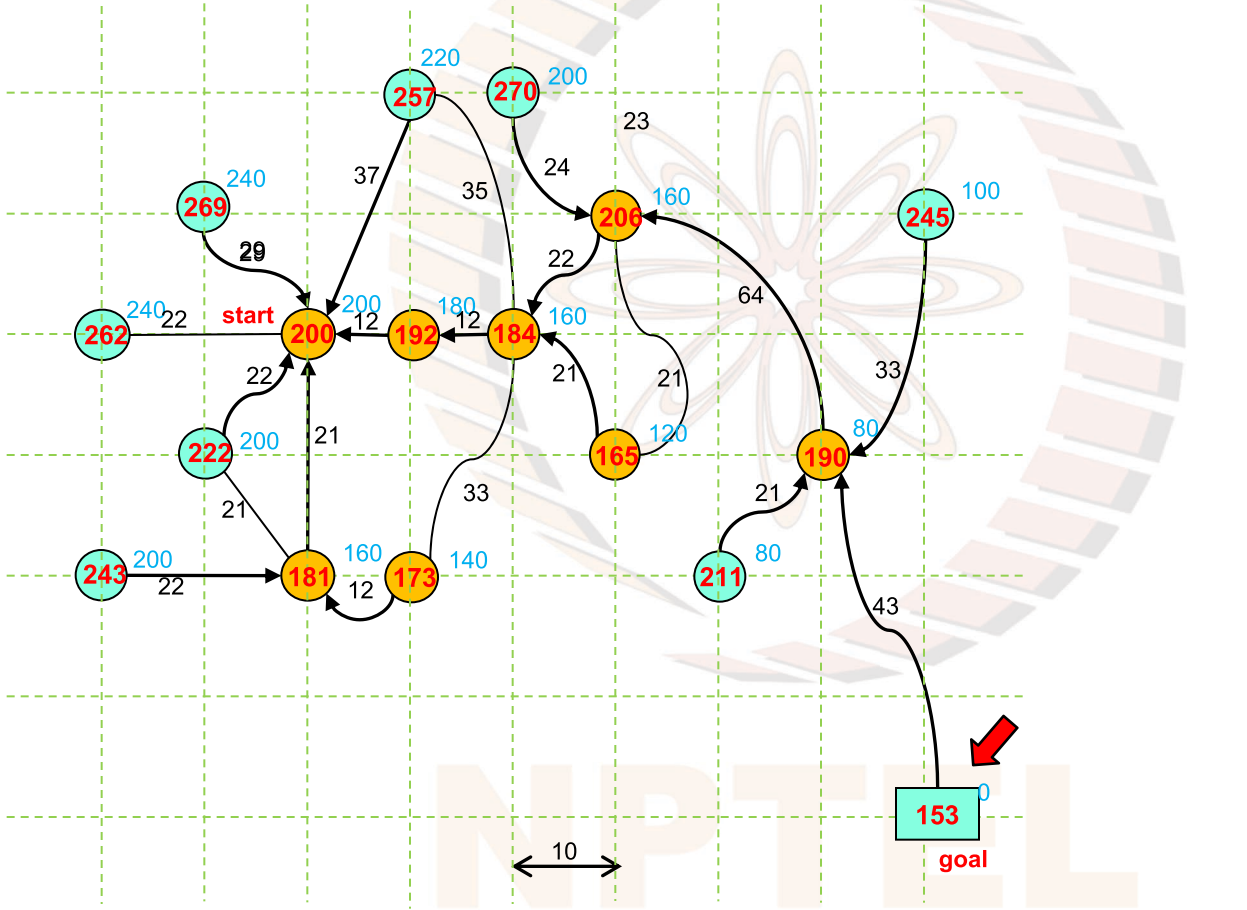


Weighted A*: f(n) = g(n) + 2 × h(n)

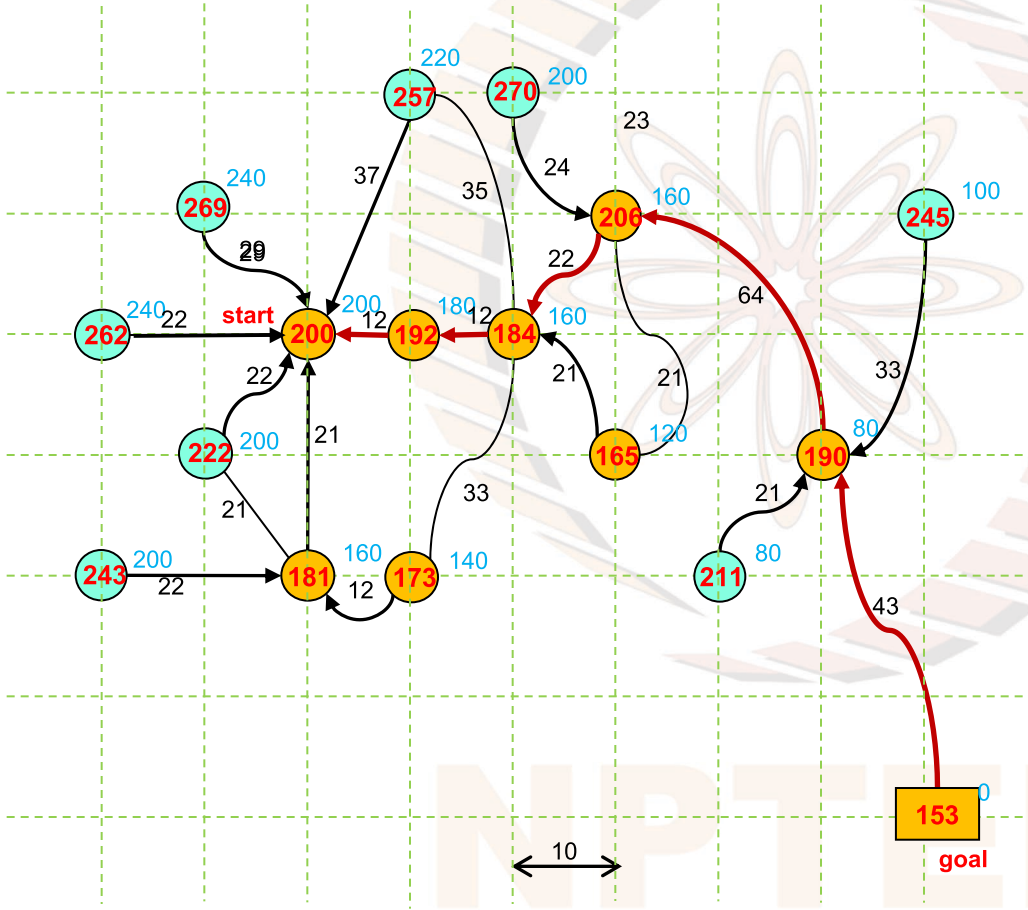


The rapidly decreasing weighted heuristic pushes the search faster towards the goal, exploring less of the state space than the A* algorithm.

Weighted A8 finds the goal faster than A*



Weighted A*: f(n) = g(n) + 2 × h(n)

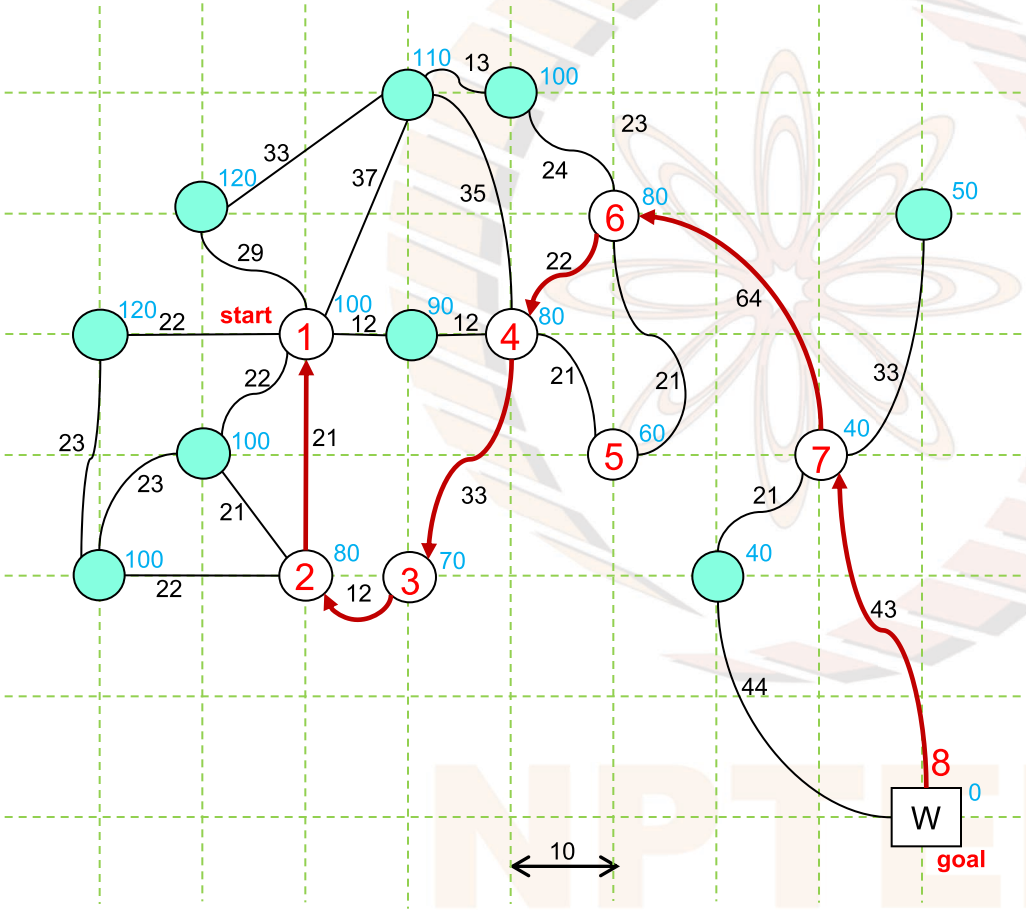


Weighted A* finds a path to the goal after inspecting 9 nodes.

But it finds a more expensive path to the goal than A*.

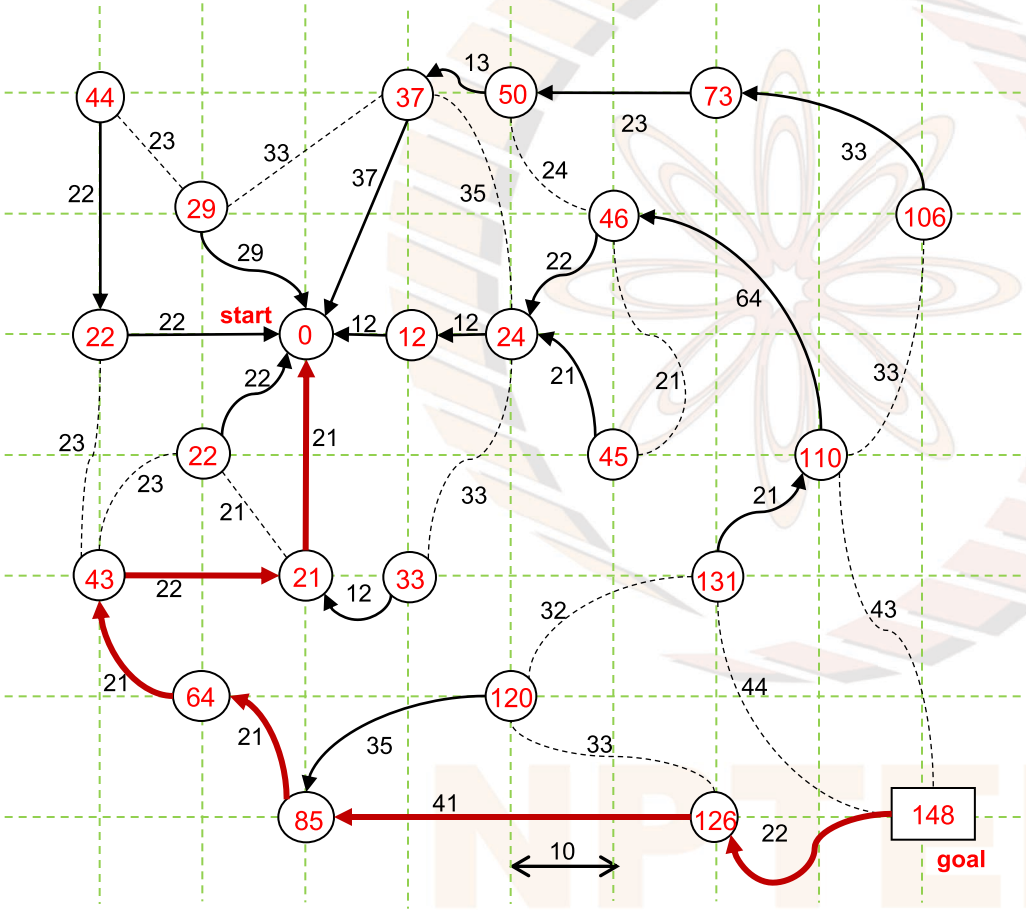
Cost of path = 153

Path found by Best First Search



8 nodes inspected
Cost of Solution = 195

Branch & Bound : f(n) = g(n) + 0 × h(n)

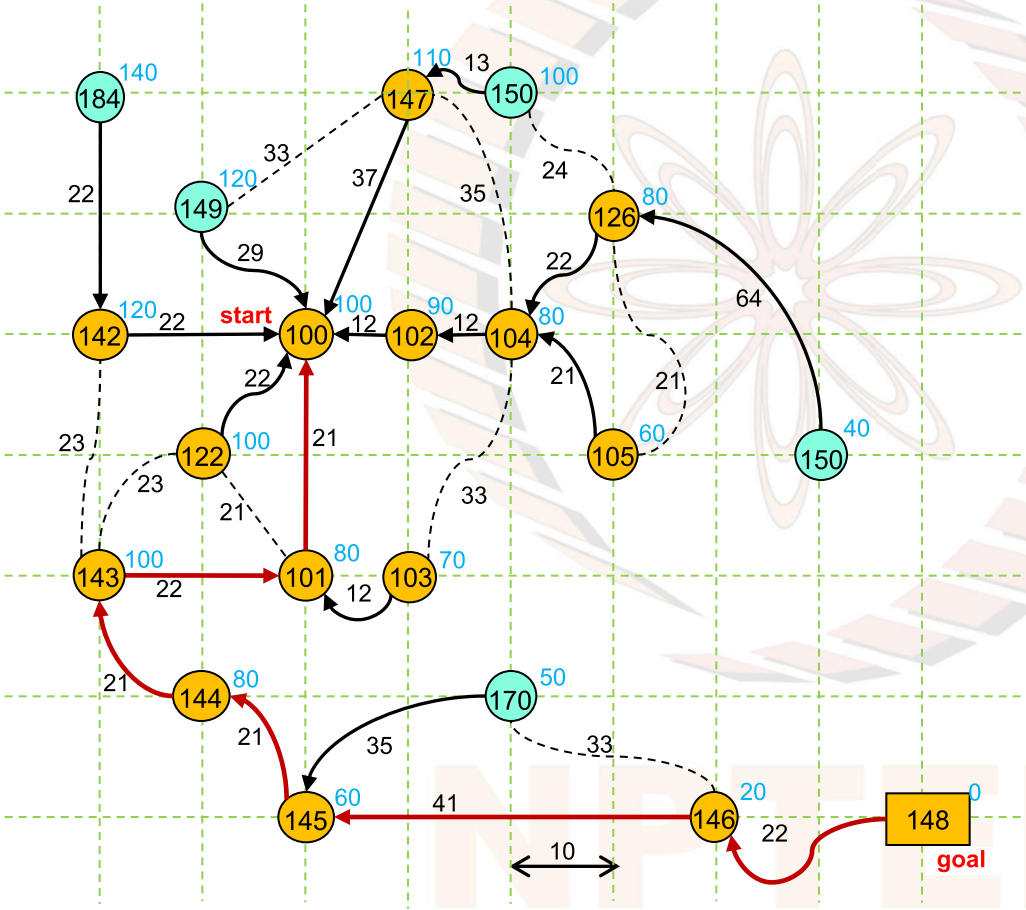


B&B explores the *entire* graph before finding the *optimal path* to the goal node.

No looking ahead!

Nodes are labeled with g*(n) values.

A*: The Solution



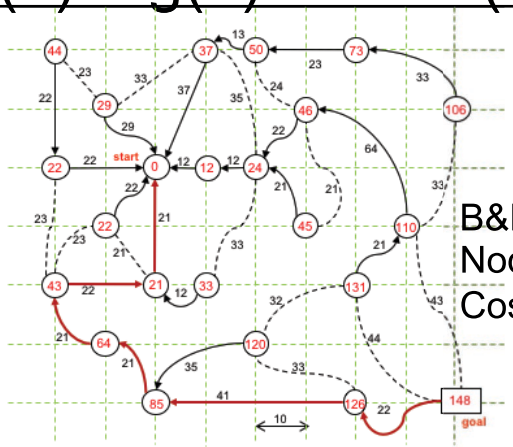
14 nodes inspected
Cost of solution = 148

$f(n) = g(n) + w \times h(n)$: B&B - A* - wA* - Best First

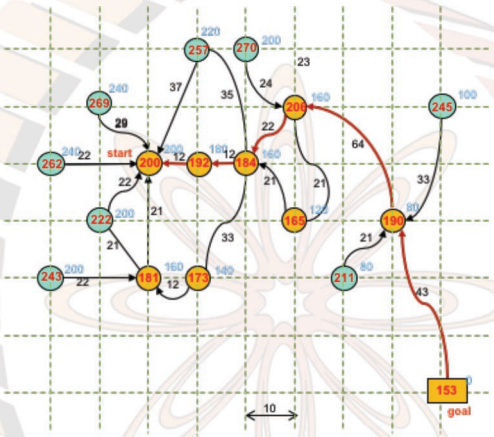
Weight w	Algorithm	Nodes generated	Nodes inspected	Cost of path
0	Branch & Bound	23	23	148
1	A*	19	14	148
2	Weighted A*	17	9	153
∞^1	Best First Search	17	8	195

Note[1] : This is equivalent to saying that
the weight of g(n) is zero, that is, negligible

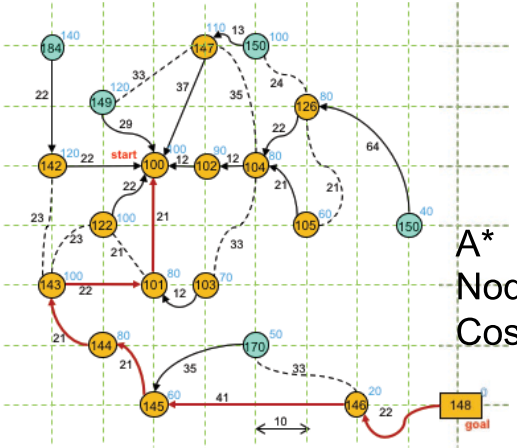
f(n) = g(n) + w × h(n): B&B - A* - wA* - Best First



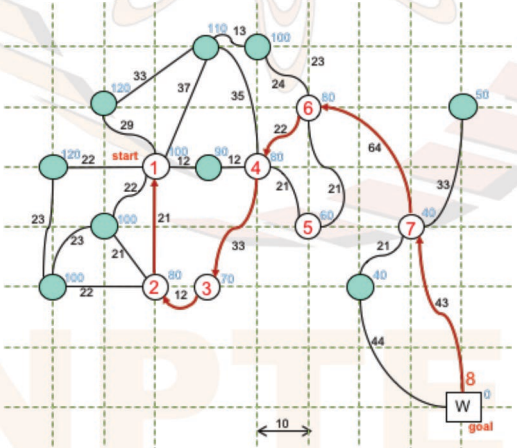
B&B
Nodes 23
Cost 148



wA*
Nodes 9
Cost 153



A*
Nodes 14
Cost 148



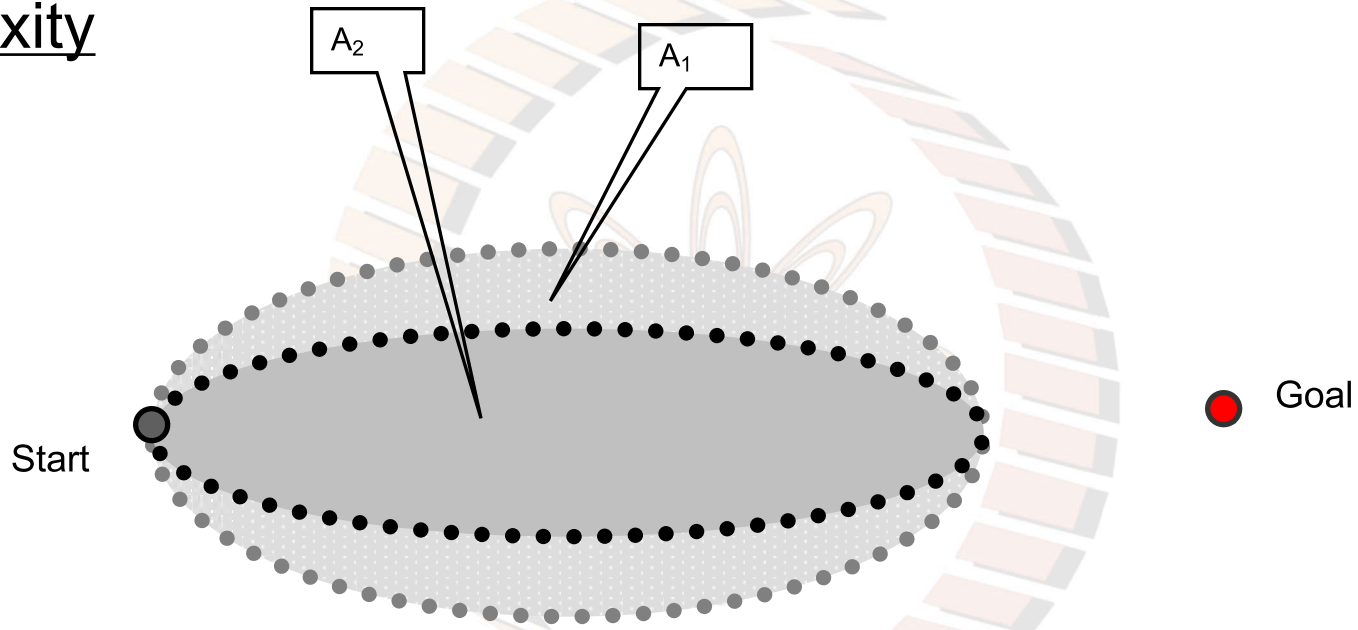
Best First
Nodes 8
Cost 195

Next

Admissible Variations on A*

Leaner, meaner versions...

Complexity

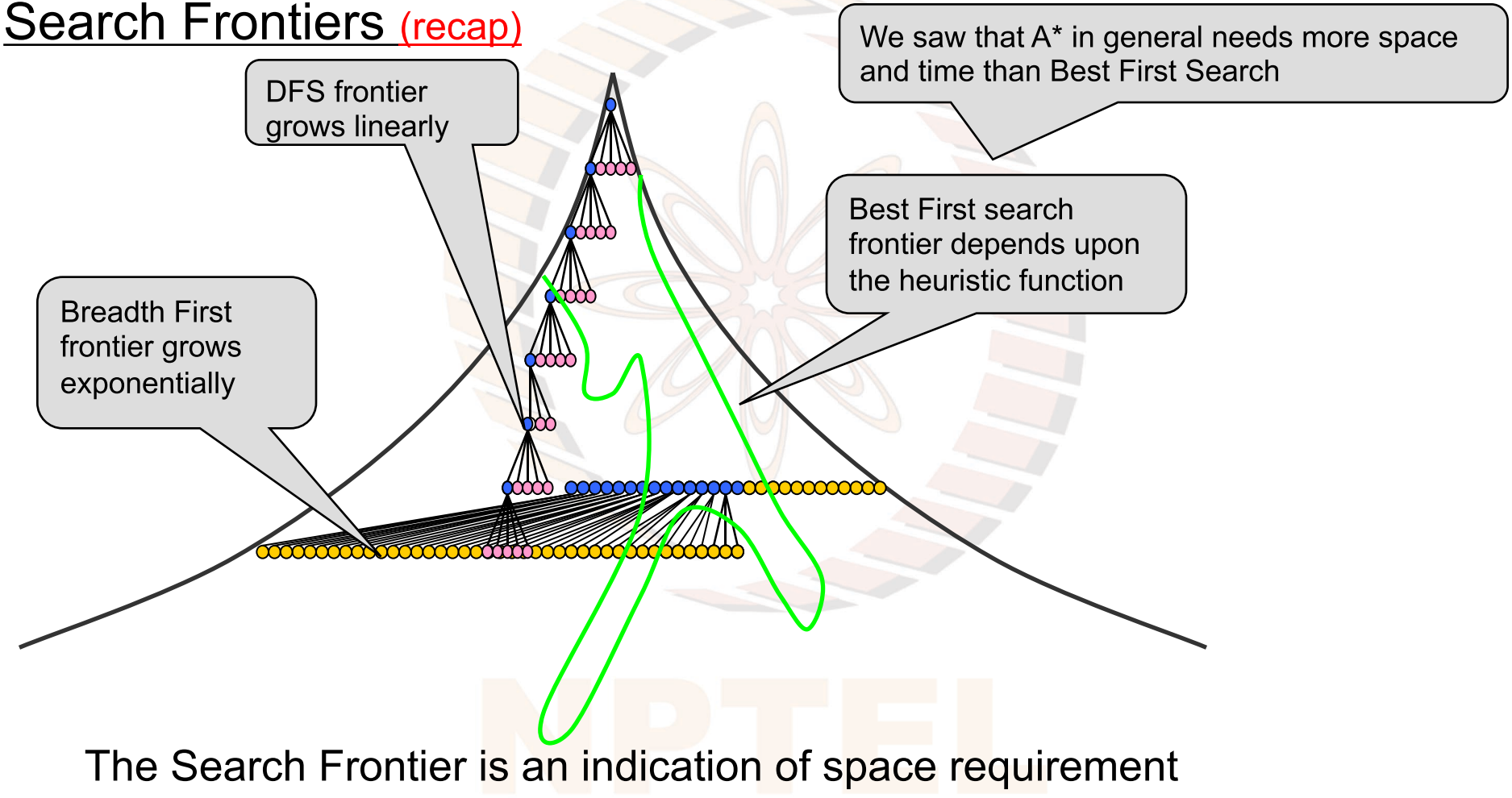


The search spaces for A_1 and A_2 when $h_2(n) > h_1(n)$.

Here h_2 was more informed than h_1

In weighted A* we can shrink this further but *lose admissibility*.

Search Frontiers (recap)



The Search Frontier is an indication of space requirement

Saving on space

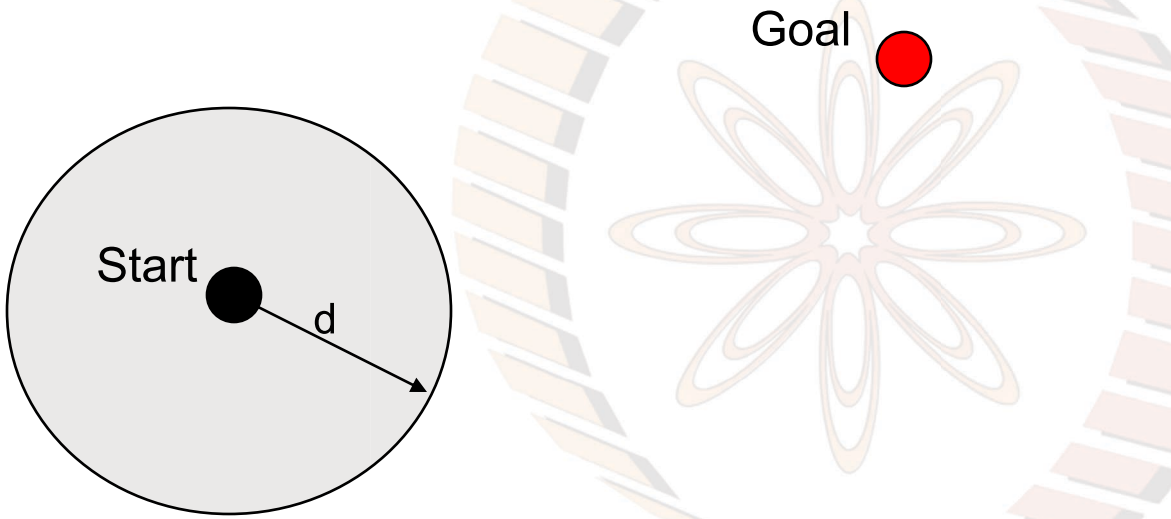
- While studying Best First Search we had observed that the space complexity of the algorithm is often exponential
  - depending upon how good the heuristic function is

- This prompted us to look at local search algorithms like Hill Climbing, Beam Search, Tabu Search and Simulated Annealing
- We also looked at population based methods like Genetic Algorithms and Ant Colony Optimization

- But these algorithms did not guarantee optimality.

- Are there space saving versions of A* that are admissible?
  - Perhaps at the expense of time complexity?
  - Remember Depth First Iterative Deepening?

Depth Bounded DFS (DBDFS) (recap)



Do DFS with a depth bound *d*.

Linear space

Not complete

Does not guarantee shortest path

Depth First Iterative Deepening (DFID) (recap)

```
DepthFirstIterativeDeepening(start)
    1    depthBound ← 1
    2    while TRUE
    3            do      DepthBoundedDFS(start, depthBound)
    4                    depthBound ← depthBound + 1
```

DFID does a series of *DBDFS*s with increasing depth bounds

A series of depth bounded depth first searches

(thus requiring linear space)

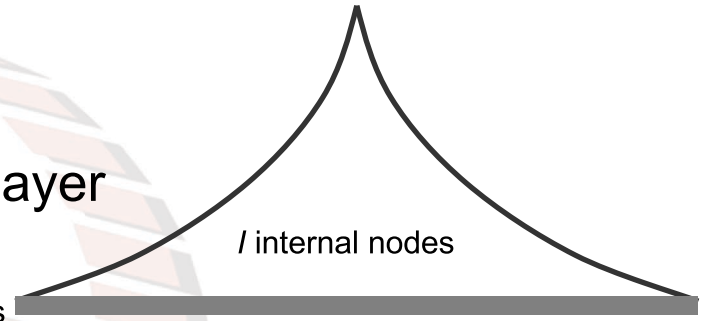
of increasing depth bound.

When a path to goal is found in some iteration it is the shortest path

(otherwise it would have been found in the previous iteration)

DFID: cost of extra work

For every new layer in the search tree that DFID explores, it searches the entire tree up to that layer all over again.



DFID inspects ($L+I$) nodes whereas
Breadth First would have inspected L nodes

$$N_{DFID} = N_{BFS} \frac{(L+I)}{L}$$

But $L = (b-1)*I + 1$ for a full tree with branching factor b

$$\therefore \quad N_{DFID} \approx N_{bFS} \frac{b}{(b-1)}$$

for large L where b is the branching factor

The extra cost is not significant!

Iterative Deepening A*

- The algorithm Iterative Deepening A* (IDA*) was presented by Richard Korf in 1985
- It is designed to save on space by doing a series of Depth First Searches of increasing depth
  - Like Depth First Iterative Deepening
- Unlike DFID which uses depth as a parameter, IDA* uses f-values to determine how far should Depth First Search go
- IDA* initially sets the bound to f(S)
  - which is equal to h(S) and is an underestimate on optimal cost
- In subsequent cycles it extends the bound to the next unexplored f-value

Iterative Deepening A* (IDA*)

```
IterativeDeepeingA*(start)
    1    depthBound ← f(S)
    2    while TRUE
    3            do      DepthBoundedDFS(start, depthBound)
    4                    depthBound ← f(N) of cheapest unexpanded node on OPEN
```

IDA* does a series of *DBDFS*s with increasing depth bounds

A series of depth bounded depth first searches

(thus requiring linear space)

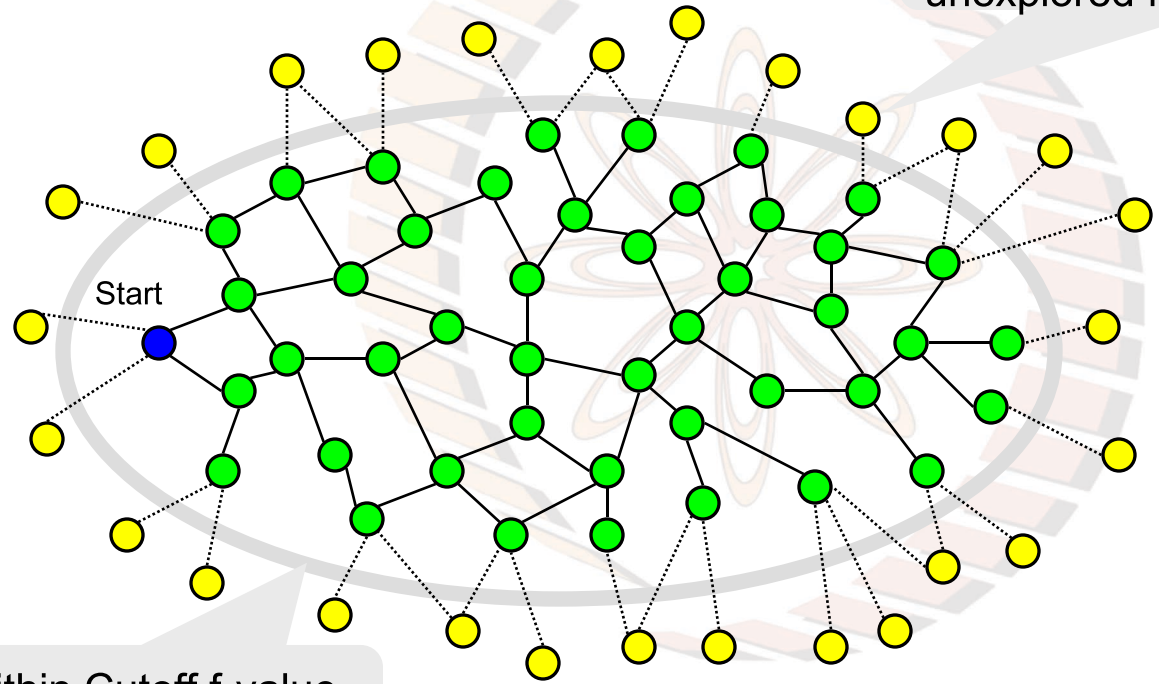
of increasing depth bound.

When a path to goal is found in some iteration it is the shortest path

(otherwise it would have been found in the previous iteration)

IDA*: nodes explored by DFS

In the next cycle cutoff to lowest unexplored f-value node



Goal

Nodes within Cutoff f-value are explored by DFS

IDA* iteratively extends search frontier for Depth First Search

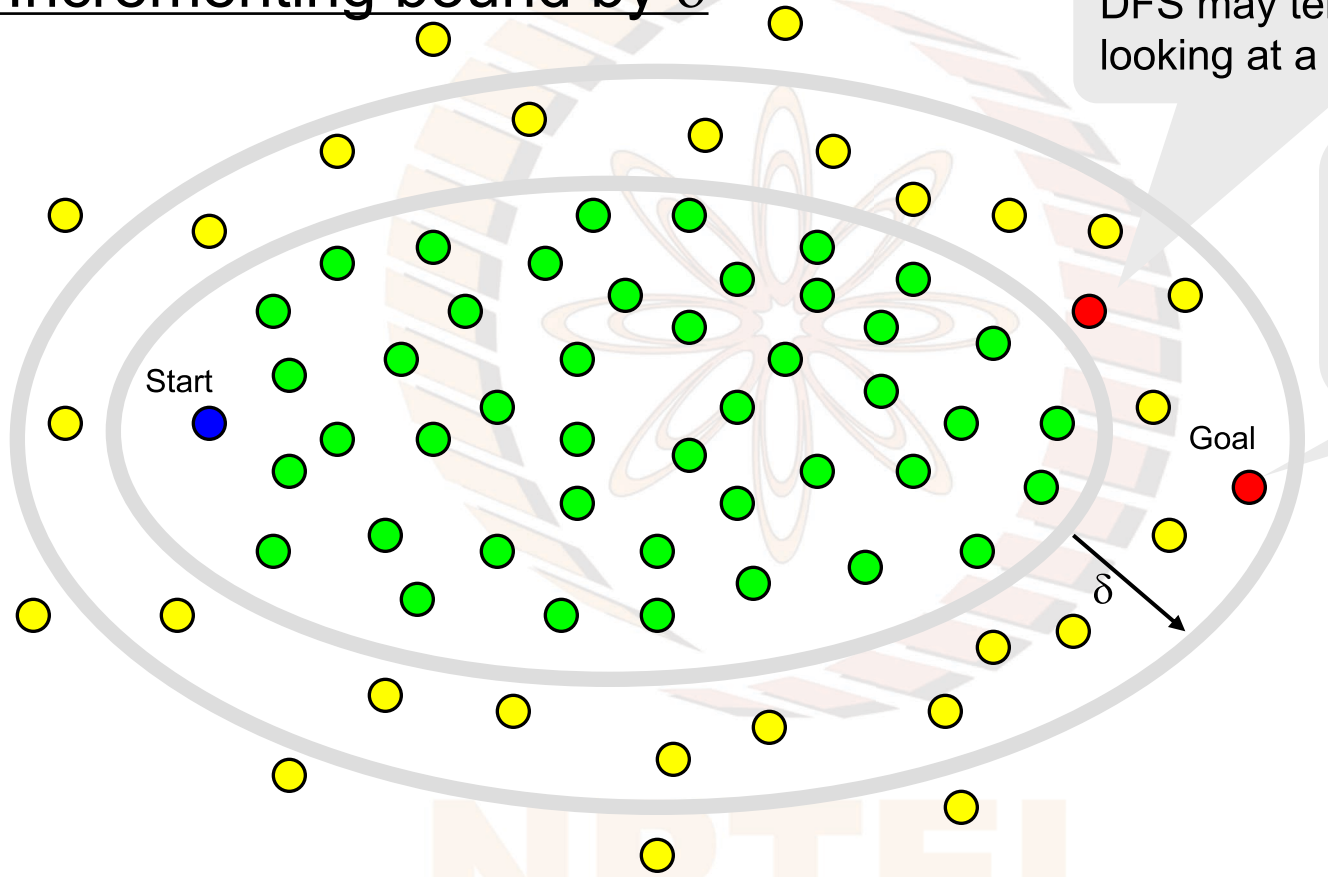
Problems with IDA*

- Even when the state space grows quadratically the the number of paths to each node grows exponentially.

- The DFS algorithm can spend a lot of time exploring these different paths if a CLOSED list is not maintained
  - We observed that this has to be done to guarantee shortest path in the case of DFID

- The other problem is that in each cycle it extends the bound only to the next f-value
  - This means that the number of cycles may become too large specially if the nodes have all different f-values
  - One option is to extend the bound by a certain constant amount each time, with a limited compromise on the cost

IDA*: Incrementing bound by δ



DFS may terminate before looking at a cheaper solution.

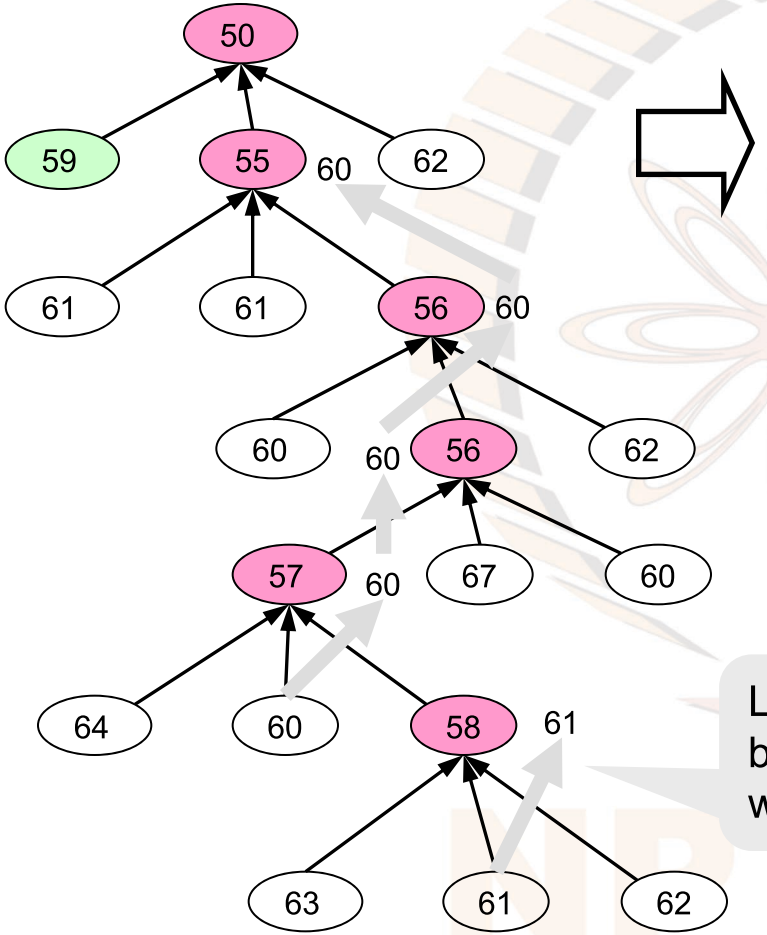
IDA* may find a costlier solution. The extra cost is bounded by δ

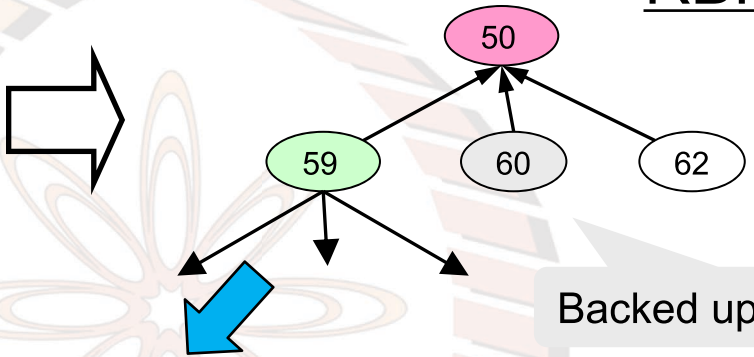
Loss in optimality is bounded by δ

Recursive Best First Search

- IDA* has linear space requirements (being Depth First in nature)
- But it has no sense of direction (again being Depth First in nature)
- Recursive Best First Search (RBFS) is an algorithm also presented by Richard Korf (1991)
- "*RBFS is a linear-space best-first search algorithm that always explores new nodes in best-first order, and expands fewer nodes than iterative deepening with a nondecreasing cost function.*"
- RBFS is like Hill Climbing with backtracking
  - Backtracking if *no child is best node* on OPEN
  - Except that instead of backtracking, RBFS *rolls back* search to a node it has *marked as second best*
  - While rolling back it *backs up the lowest value* for each node on from its children to *update* the value of the parent

RBFS: Roll back





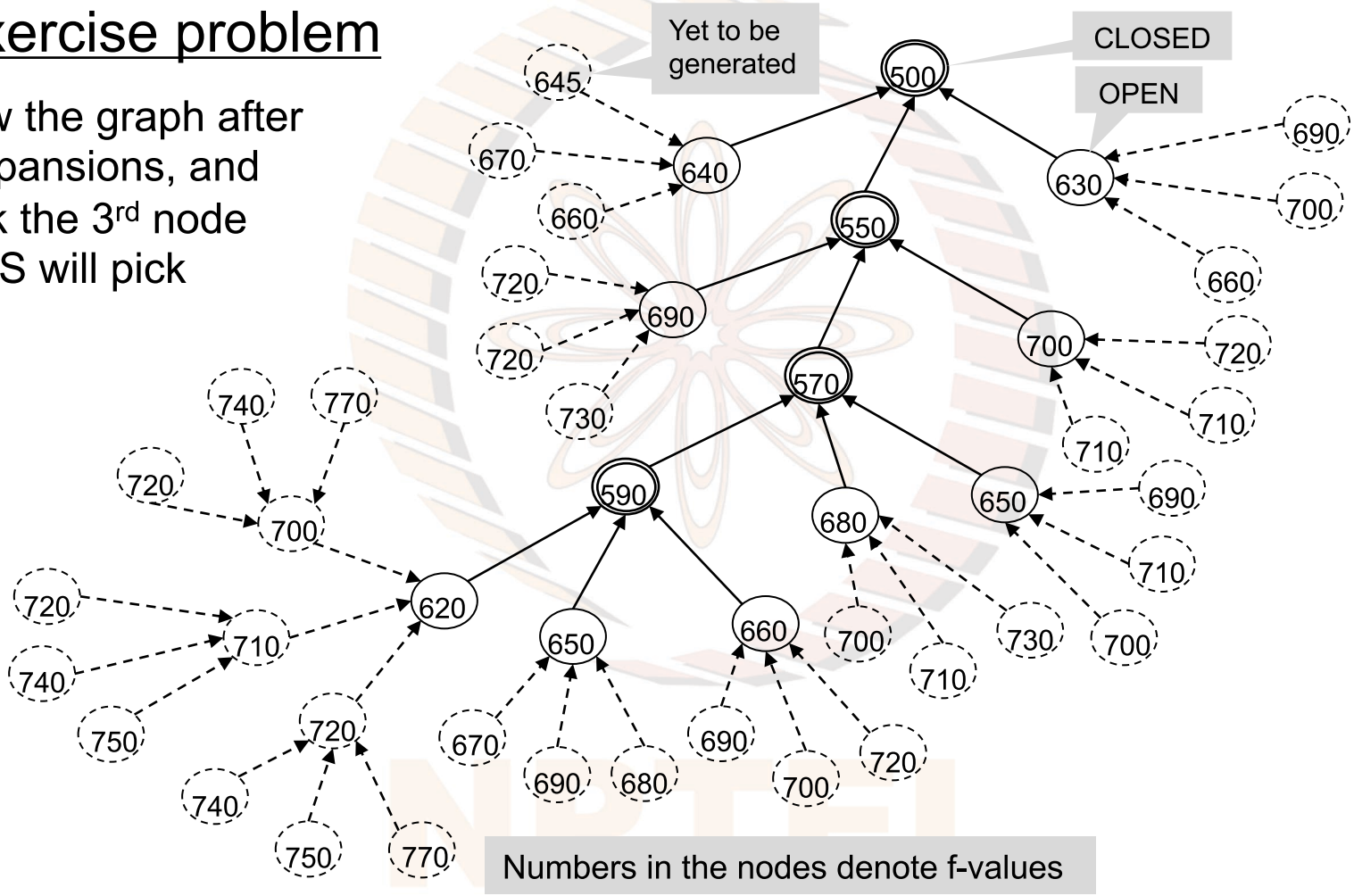
Backed up f-value

New Search

Lowest f-values are backed up at each level when RBFS rolls back

An exercise problem

Draw the graph after 2 expansions, and mark the 3rd node RBFS will pick



Next

The monotone or consistency condition

when, like Dijkstra's algorithm,

A* *too* finds the optimal path

to every node it picks from OPEN

The Monotone Condition

The *monotone property* or the *consistency property* for a heuristic function says that for a node *n* that is a *successor* to a node *m* on a path to the goal being constructed by the algorithm *A** using the heuristic function h(x),

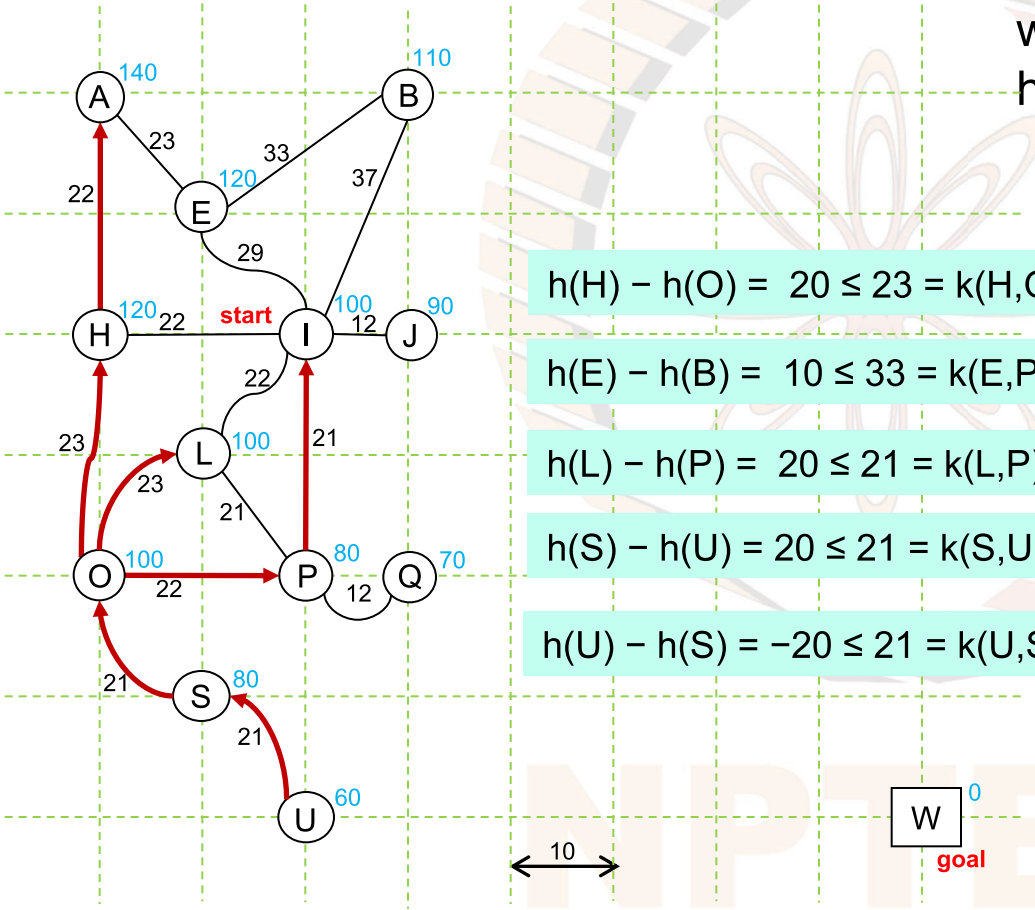
$$h(m) - h(n) \leq k(m,n)$$

This means that this is true for any two nodes M and N connected by an edge.

The heuristic function

underestimates the cost of each edge

h(M) − h(N) ≤ k(M,N)



The nodes are placed on a grid where each edge is 10km, and h(n) is the Manhattan distance

h(H) − h(O) = 20 ≤ 23 = k(H,O)

h(E) − h(B) = 10 ≤ 33 = k(E,P)

h(L) − h(P) = 20 ≤ 21 = k(L,P)

h(S) − h(U) = 20 ≤ 21 = k(S,U)

h(U) − h(S) = −20 ≤ 21 = k(U,S)

Even on an optimal path to the goal the difference in h-values is less than the edge cost

Monontone condition → non-decreasing f-values

Given that we started from start *S,* we can add the term g(*m*) to both sides to get,

$$h(m) + g(m) \leq k(m,n) + h(n) + g(m)$$

Since *n* is the successor of *m* we have

$$g(m) + k(m,n) = g(n)$$

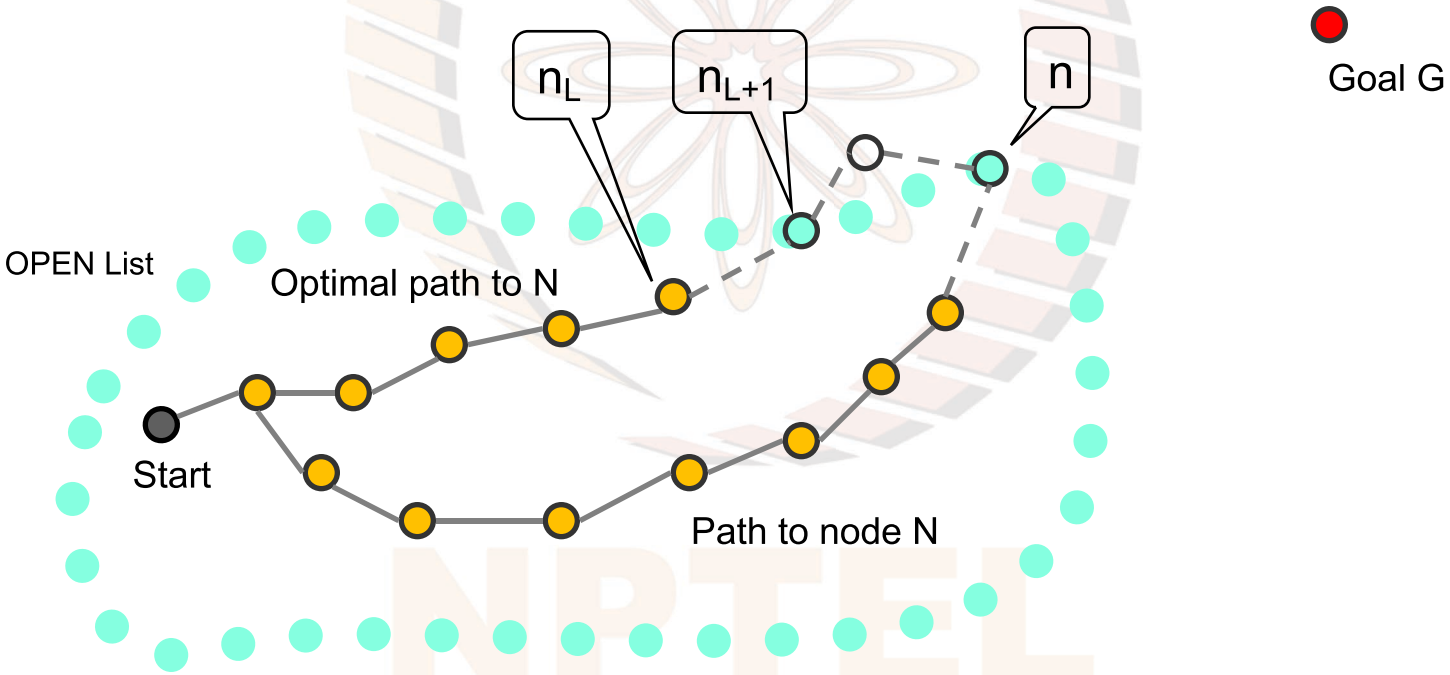
Therefore

$$h(m) + g(m) \leq h(n) + g(n),$$

or $$f(m) \leq f(n)$$

L7: Monotone condition → when *A** picks a node *n*, g(n) = g*(n)

Proof: Let *A** expand node *n* with cost g(*n*).
Let n_L be the last node on the *optimal path* from *S* to *n* that has been expanded. Let n_{L+1} be the successor of n_L that must be on *OPEN*.



Proof (continued): g(n) = g*(n)

Given that $f(n_L) \leq f(n_{L+1})$ the following property holds,

$$h(n_L) + g(n_L) \leq h(n_{L+l}) + g(n_{L+l})$$

or $h(n_L) + g^*(n_L) \leq h(n_{L+l}) + g^*(n_{L+l})$ because both are on the optimal path

By transitivity of ≤ the above property holds true for *any two nodes* on the optimal path. In particular it holds for n_{L+l} and node *n*. That is,

$$h(n_{L+l}) + g^*(n_{L+l}) \leq h(n) + g^*(n)$$

That is,

$$f(n_{L+l}) \leq h(n) + g^*(n)$$

because n_{L+1} is on the optimal path to *n*

Proof (continued): g(n) = g*(n)

But since *A** is about to pick node *n* instead,

$f(n) \leq f(n_{L+1})$

That is,

$h(n) + g(n) \leq f(n_{L+1})$

or $h(n) + g(n) \leq h(n_{L+1}) + g^*(n_{L+1})$

Recall that $h(n_{L+1}) + g^*(n_{L+1}) \leq h(n) + g^*(n)$

Therefore

$h(n) + g(n) \leq h(n) + g^*(n)$

or $g(n) \leq g^*(n)$

or $g(n) = g^*(n)$

because g(*n*) cannot be less than g*(*n*) the optimal cost.

The consequences: CLOSED is closed

For *A** the interesting consequence of searching with a heuristic function satisfying the monotone property is that every time it picks a node for expansion, it has found an optimal path to that node.

As a result, there is no necessity of improved cost propagation through nodes in *CLOSED* (Case 3 in A*),

...because *A** would have already found the best path to them in the first place when it picked them from *OPEN* and put them in *CLOSED*.

Next

We look at a problem in which pruning the CLOSED would be desirable:

Sequence Alignment in Biology

DNA sequences

A nucleic acid sequence is a succession of letters that indicate the order of nucleotides forming alleles within a DNA (using GACT) or RNA (GACU) molecule.

Because nucleic acids are normally linear (unbranched) polymers, specifying the sequence is equivalent to defining the covalent structure of the entire molecule.

The sequence has capacity to represent information. Biological deoxyribonucleic acid represents the information which directs the functions of a living thing.

The possible letters are A, C, G, and T, representing the four nucleotide bases of a DNA strand — adenine, cytosine, guanine, thymine

The Sequence Alignment problem

The Sequence Alignment problem is one of the fundamental problems of *biological sciences*, aimed at finding the similarity of two amino-acid sequences.

Comparing amino-acids is of prime importance to humans, since it gives vital information on evolution and development.

Saul B. Needleman and Christian D. Wunsch devised a dynamic programming algorithm to the problem and got it published in 1970.

Since then, numerous improvements have been made to improve the time complexity and space complexity,

https://www.geeksforgeeks.org/sequence-alignment-problem/

Cost of alignment

Given two sequences composed of the characters C, A, G and T the task of sequence alignment is to list the two alongside with the option of inserting a gap in either sequence.

The objective is to maximize the similarity between the resulting two sequences with gaps possibility inserted.

The similarity can be quantified by associating a cost with misalignment. Typically two kinds of penalties/costs are involved –

- *Mismatch* : if character X is aligned to a different character Y
  - The cost/penalty could be combination specific

- *Indel* : associated with inserting a gap

Best alignment depends on penalties

https://www.geeksforgeeks.org/sequence-alignment-problem/

indel = 3, mismatch = 7 total penalty = 6

X = C G
Y = C A

X = C G _
Y = C _ A

indel = 3, mismatch = 5 total penalty = 5

X = C G
Y = C A

X = C G
Y = C A

indel = 3, mismatch = 2 total penalty = 5

X = AGGGCT
Y = AGGCA

X = AGGGCT
Y = A _GGCA

The two strings need not be of equal length

Similarity functions

A similarity function is a kind of *inverse* of the distance function.

Using a similarity function transforms the sequence alignment into a *maximization* problem

Some simple similarity functions

Match: +1
Mismatch or Indel: −1

Match = 0
Indel = −1
Mismatch = −10

A fine grained similarity function

Highest weight to aligning A with A
No penalty for aligning T with C
Other mismatches have negative weight

	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

The alignment:

AGACTAGTTAC
CGA - - - GACGT

with a indel penalty of -5, would have the following score:

$S(A,C)+S(G,G)+S(A,A)+(3 \times d)+S(G,G)+S(T,A)+S(T,C)+S(A,G)+S(C,T)$

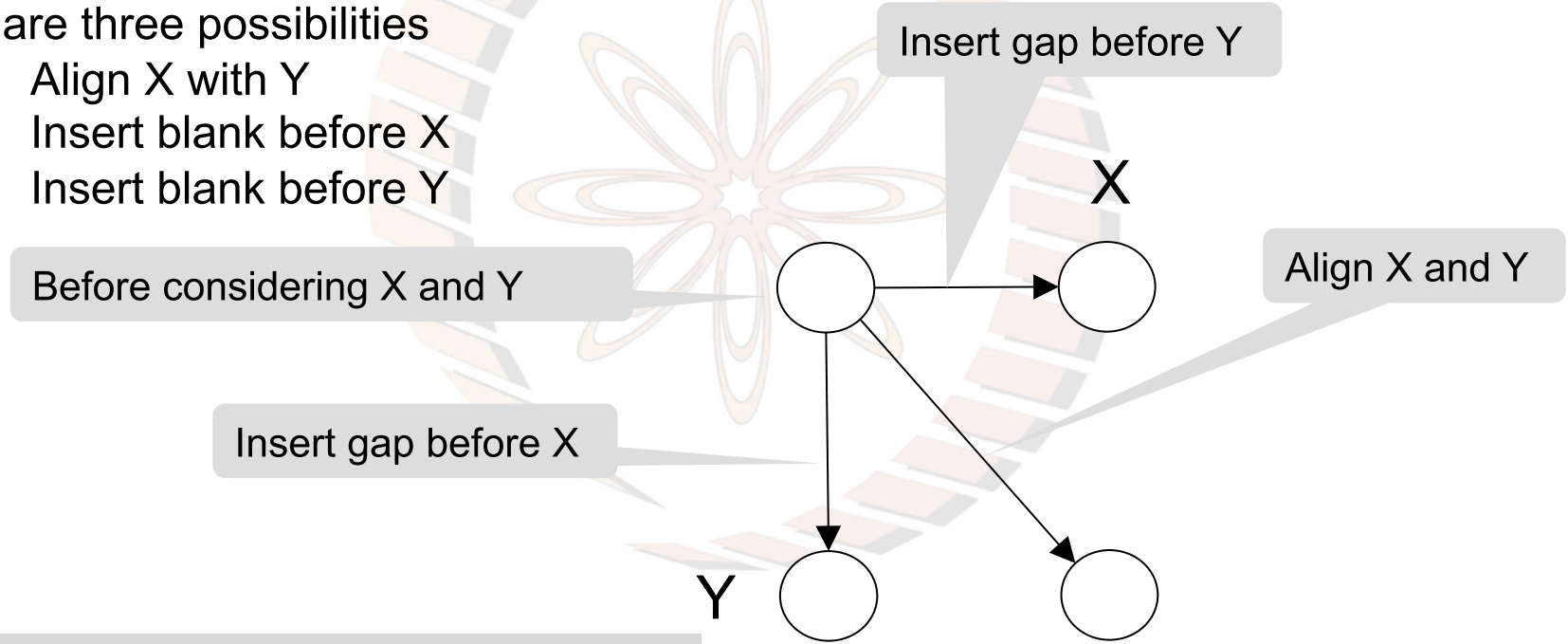
= -3 + 7 + 10 - (3 × 5) + 7 + (-4) + 0 + (-1) + 0

= 1

Sequence alignment as graph search

Let X and Y be the first characters of the two strings.
There are three possibilities

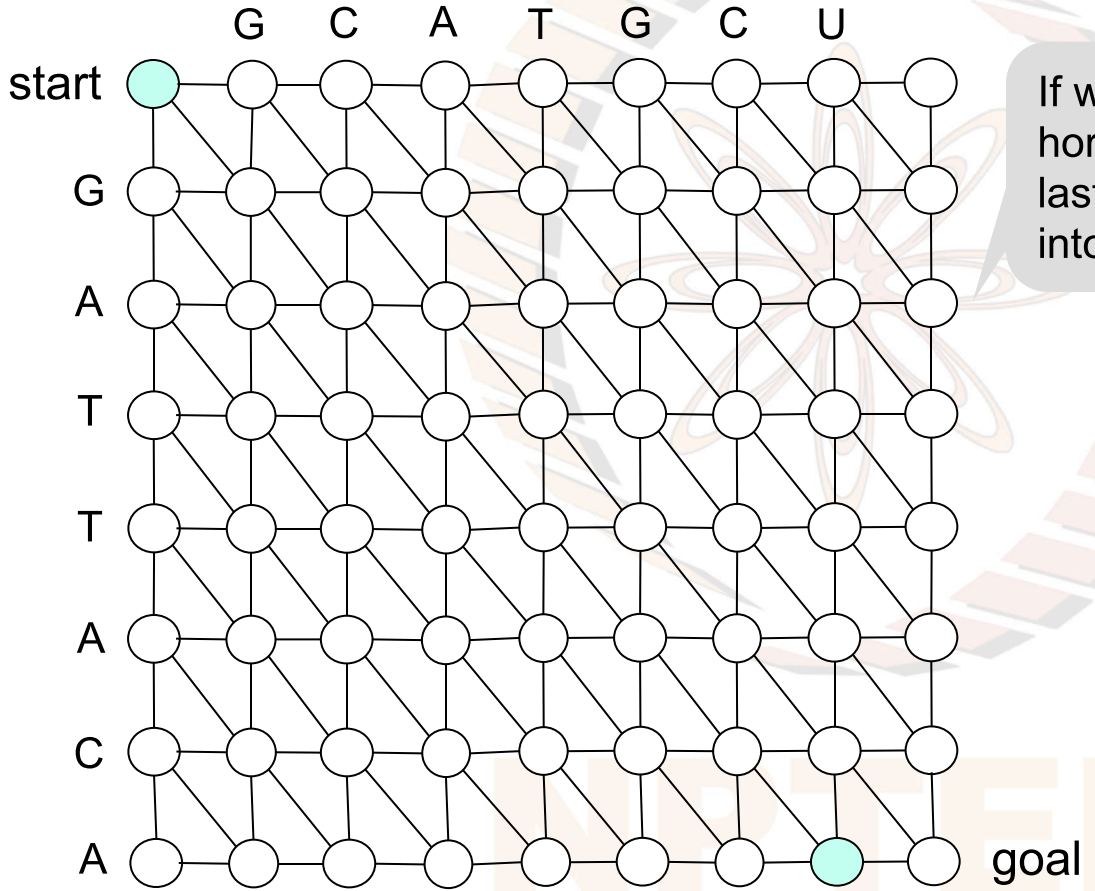
1. Align X with Y
2. Insert blank before X
3. Insert blank before Y



There are three moves from each node

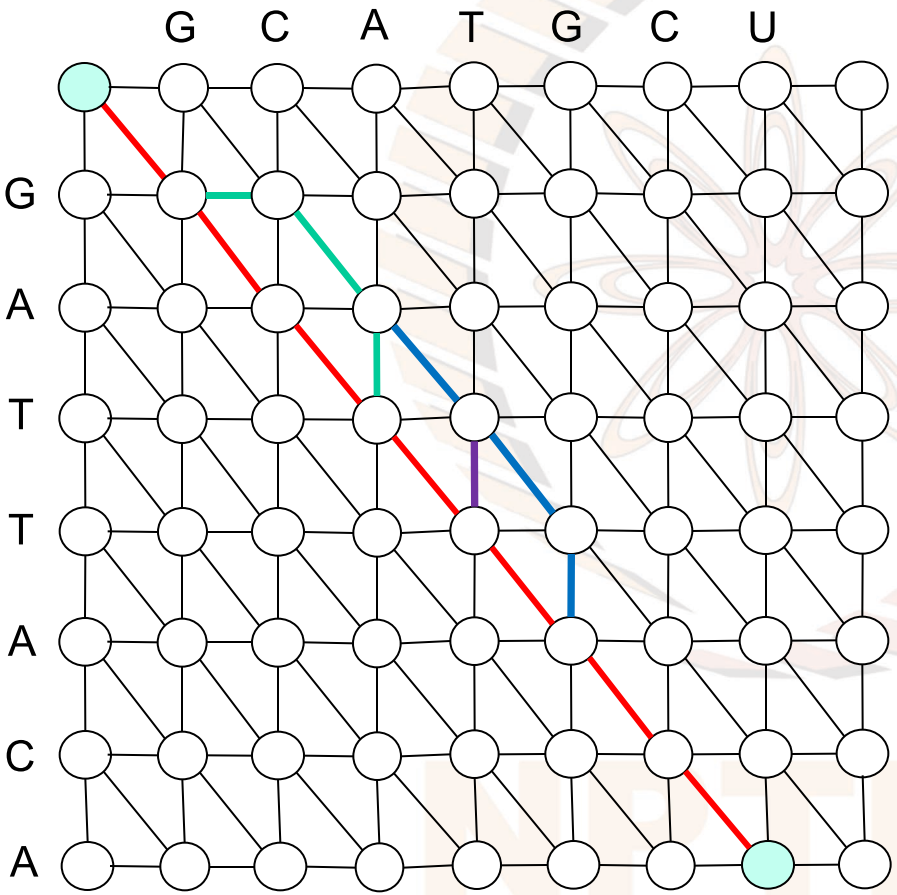
Each move has an associated penalty or cost (mismatch or indel)

The graph for the sequence alignment problem



If we had a longer horizontal sequence the last column would come into play

Four alignments



GCATGCU
GATTACA

GCATG -CU
G - ATTACA

GCA - TGCU
G - ATTACA

GCAT-GCU
G -ATTACA

The Needleman-Wunsch algorithm explores the entire space

Complexity for A*

The state space grows quadratically with depth

But the number of distinct paths grows combinatorically

Consider two strings of length N and M being aligned.
The grid size is $(N+1) \times (M+1)$

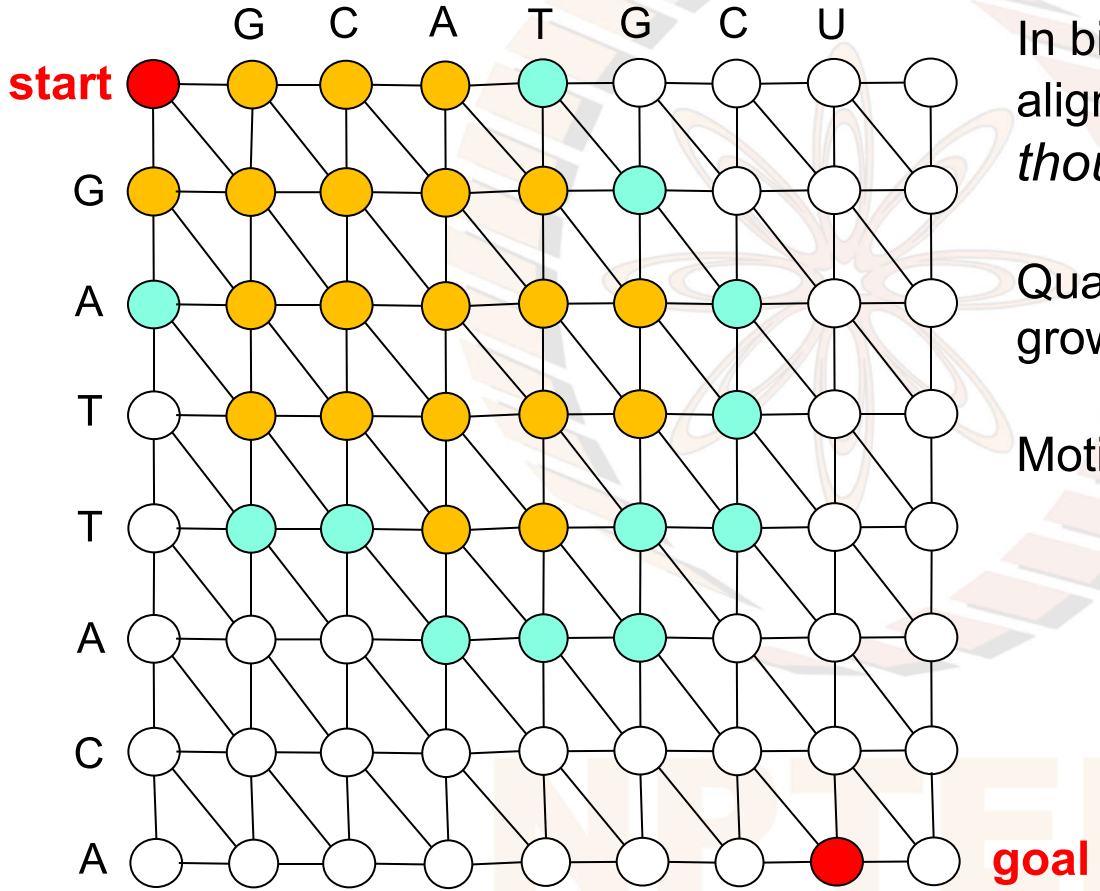
The number of ways that gaps can be inserted (*moving only horizontally or vertically*) is $(N+M)! / (N! \times M!)$

Taking *diagonal moves also* into account the number of paths is

$$\sum (M+N-R)! / (M-R)! \times (N-R)! \times R!$$

where R varies from 0 to min(M,N)
and stands for the number of diagonal moves in the path

OPEN grows Linearly, CLOSED quadratically



In biology the sequences to be aligned may have *hundreds of thousands* of characters.

Quadratic is then a formidable growth rate.

Motivation to prune CLOSED

Next

Pruning CLOSED