Artificial Intelligence:
Search Methods for Problem Solving

Algorithm A*

A First Course in Artificial Intelligence: Chapter 5

Deepak Khemani

Department of Computer Science & Engineering

IIT Madras

Algorithm A*

- Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968.

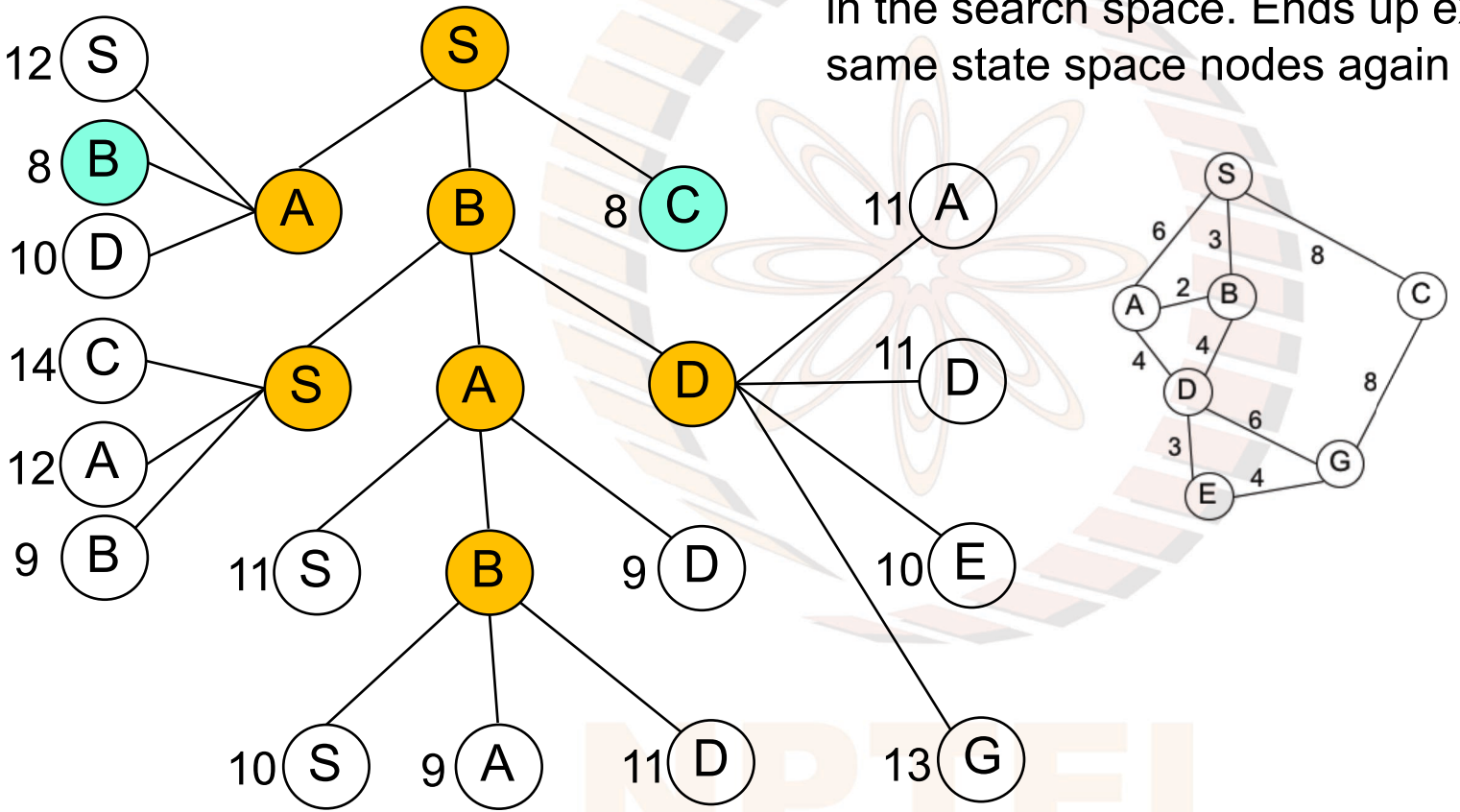
- It can be seen as an extension of Edsger Dijkstra's 1959 algorithm.

- A* achieves better performance by using heuristics to guide its search.

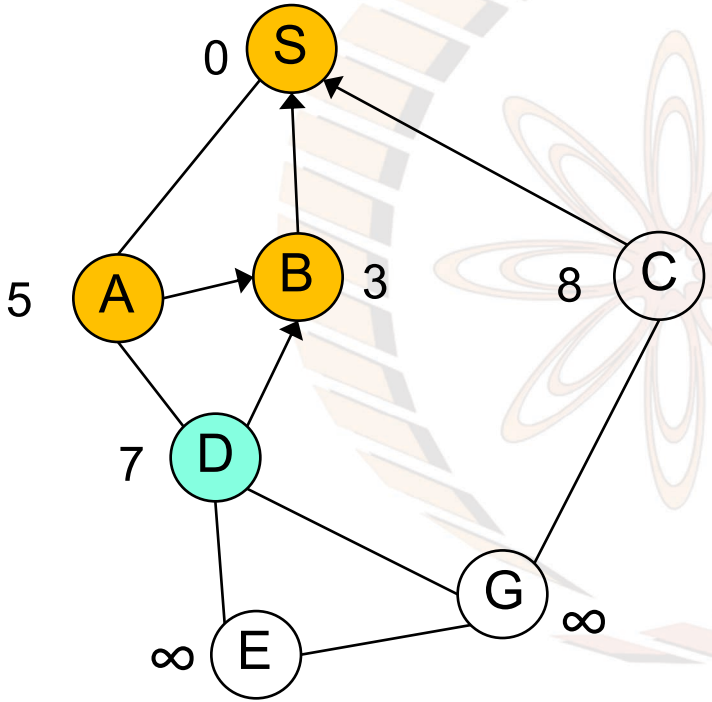
- Wikipedia

Branch & Bound (recap)

Maintains distinct paths as separate nodes in the search space. Ends up expanding same state space nodes again and again.



Dijkstra's Algorithm (recap)



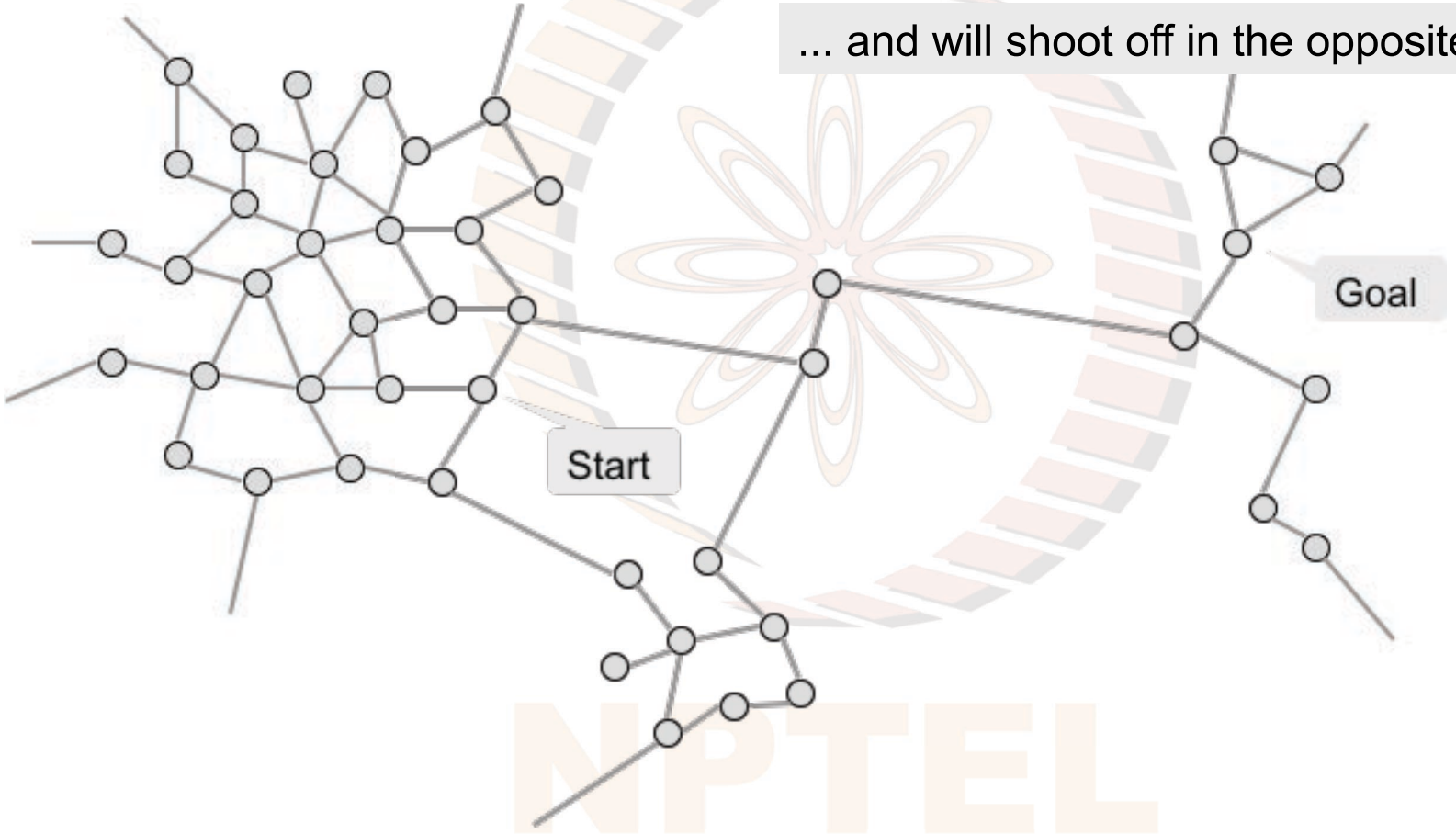
Keeps only one copy of each node, and adjusts the parent pointer when it finds a better path to a node.

Feature adopted by A*

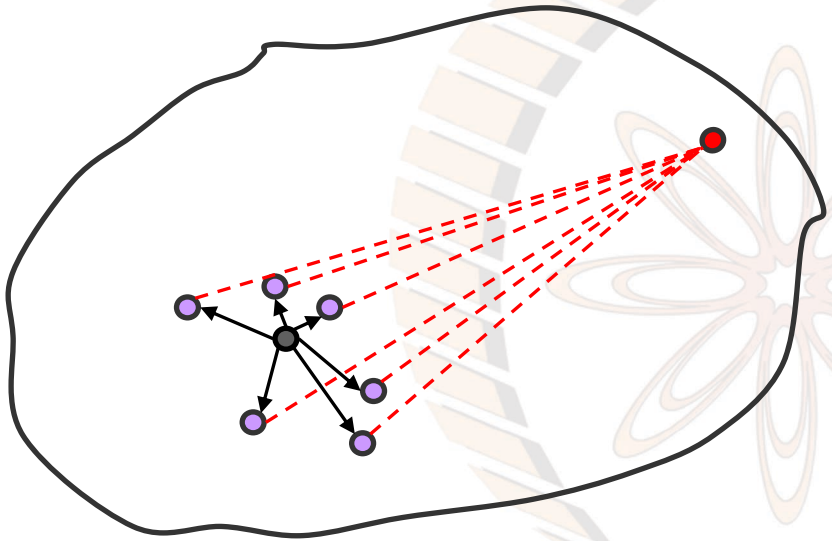
Initialize Start cost to zero, and rest to infinity
Colour cheapest node and relax the connected edges
Has no sense of direction

Both have no sense of direction..... (recap)

... and will shoot off in the opposite direction!



Best First Search (recap)



A heuristic function **estimates** the distance to the goal.

This estimate, h(n), is used to decide **which** node to pick from OPEN

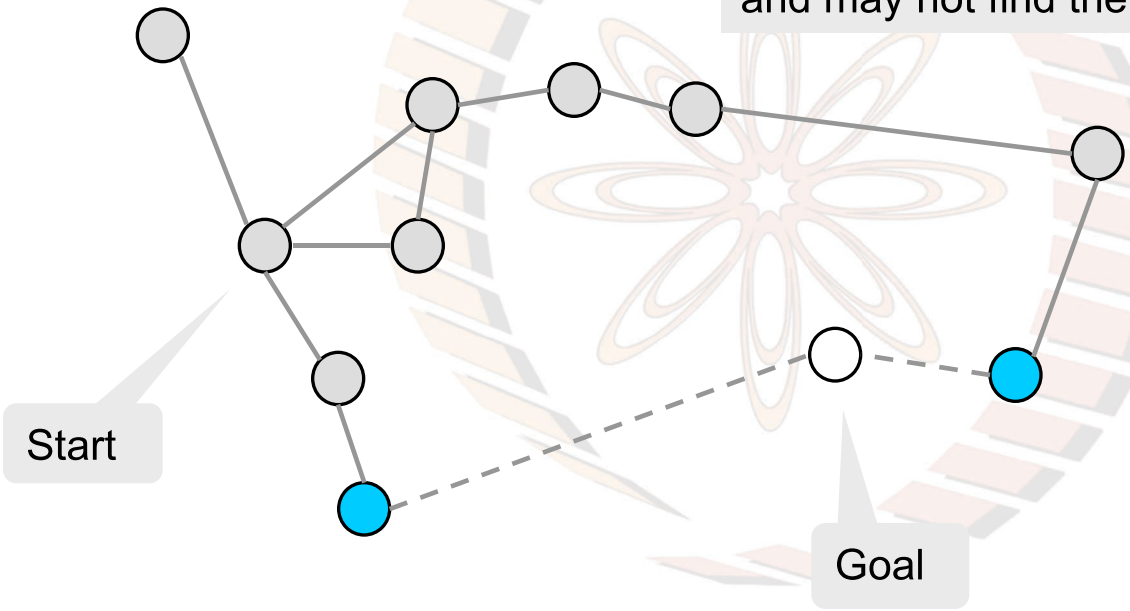
Best First Search

Candidates that appear to be *closest to the goal* are best

Chooses the candidate with the *lowest h-value node* in the hope of finding a solution *soone*r.

Best First only looks ahead (recap)

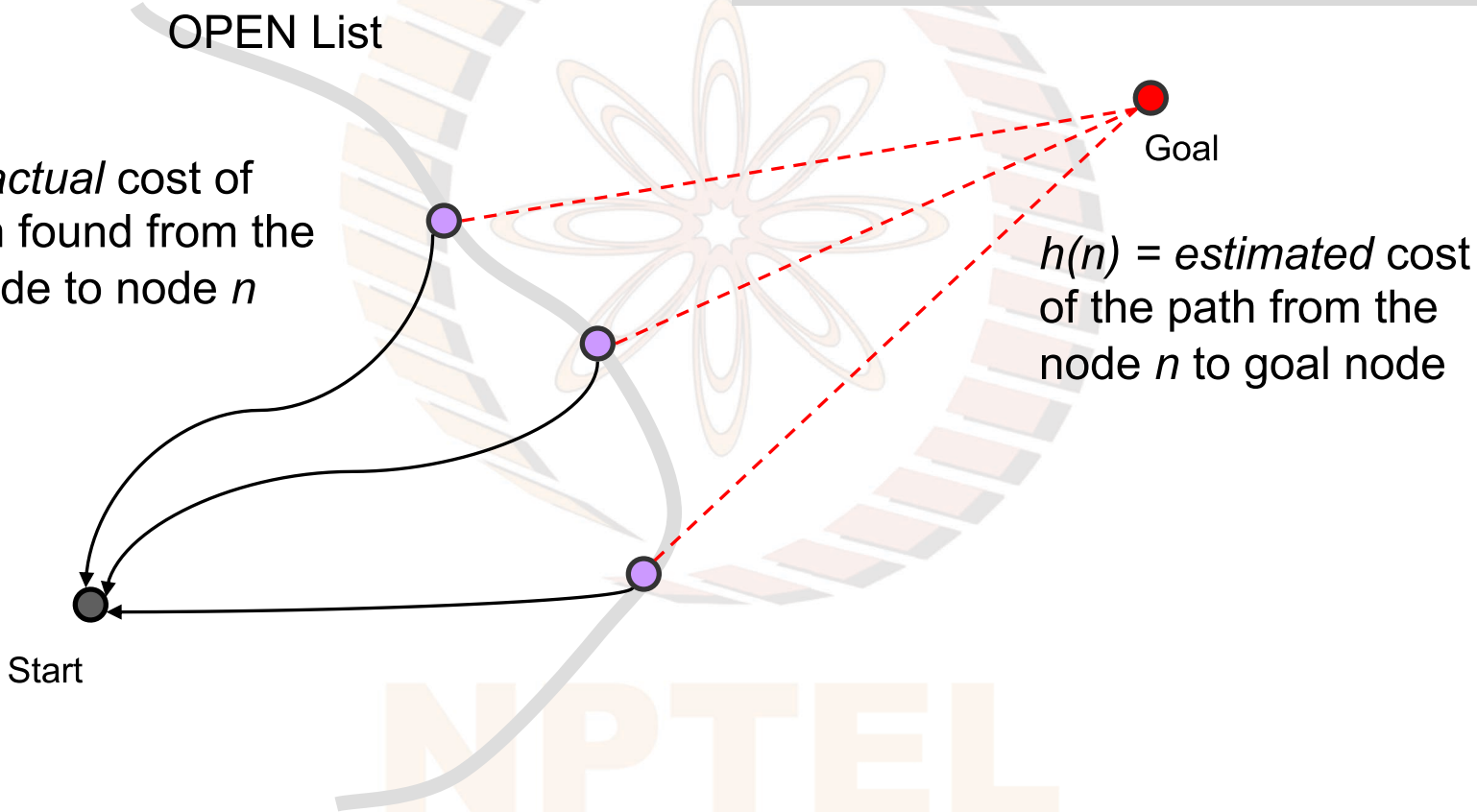
Best First only looks at the distance to goal, and may not find the shortest path



Algorithm A* combines the best features of all three algorithms!

Cost of Solution in A*

For all nodes function *f(n)* is made up of two components *g(n)* and *h(n)*



g(n) = *actual* cost of solution found from the start node to node *n*

h(n) = *estimated* cost of the path from the node *n* to goal node

Algorithm A*

Each candidate is tagged with an *estimated cost of the complete solution f(n)*,

$$f(n) = g(n) + h(n)$$

where,

g(n) is the *known* cost of path found from Start to node *n*

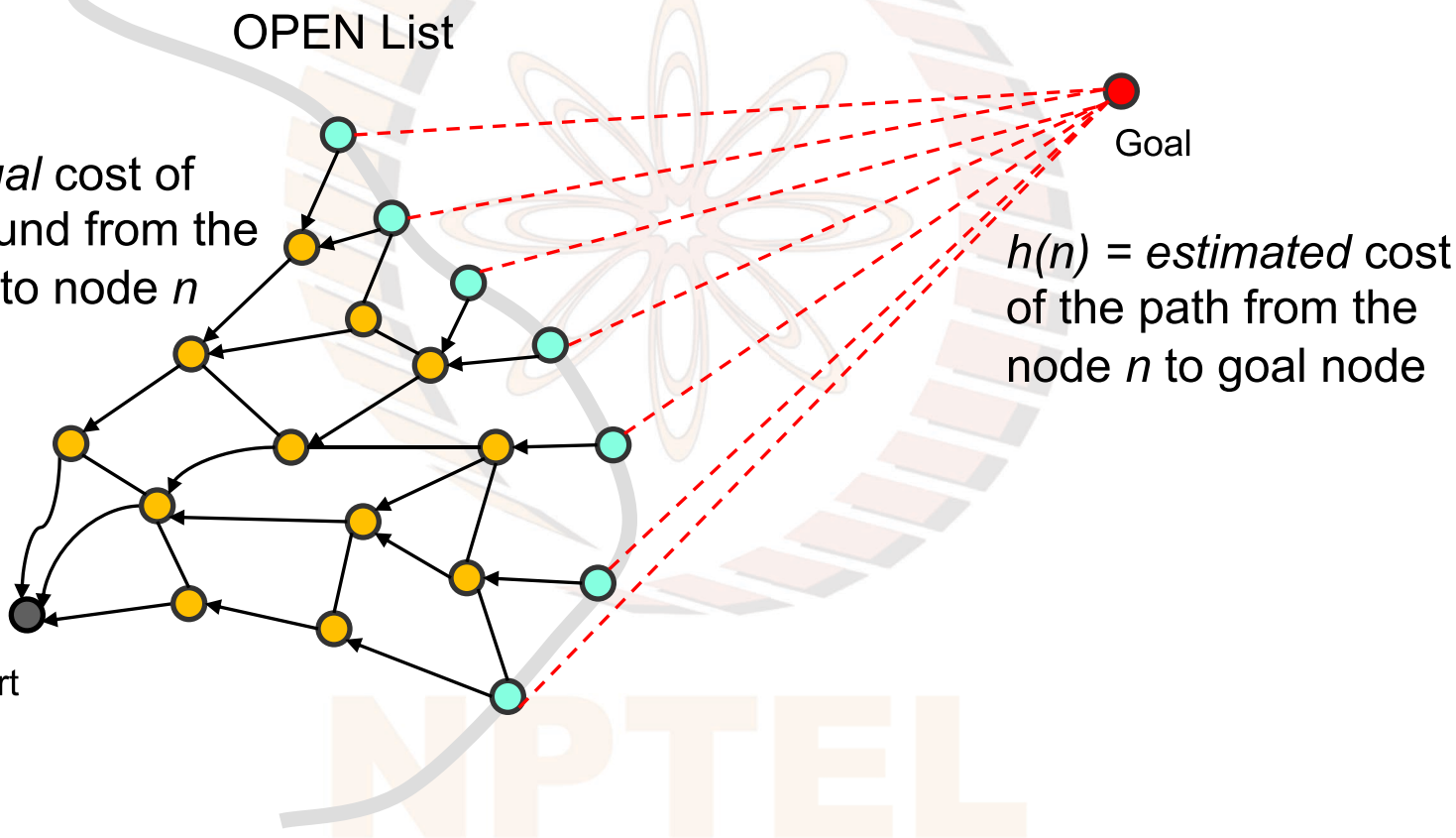
- as used by the Branch & Bound algorithm

and *h(n)* is the *estimated* cost from node *n* to the goal.

- as used by the Best First Search algorithm

The Search Space in A*

For all nodes function *f(n)* is made up of two components *g(n)* and *h(n)*

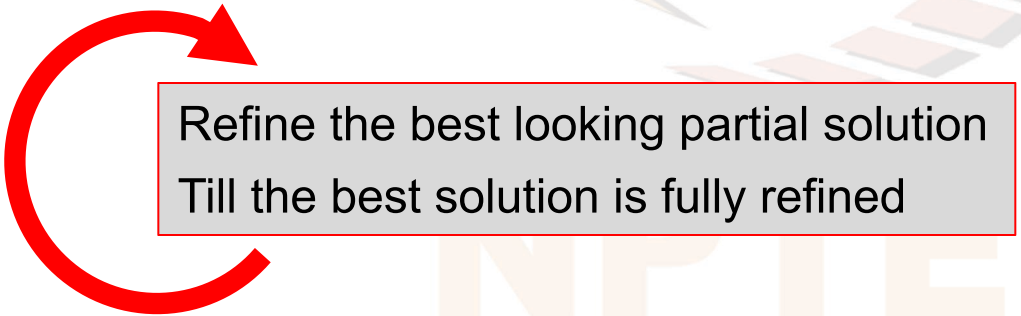


Algorithm A*

- Maintain a priority queue of nodes sorted on the *f-values*

  $f(n) = g(n) + h(n)$

- Pick the lowest f-value node, and *check* whether it is the goal
- Else generate its neighbours, and compute their *f-values*
- Insert new nodes into the priority queue
- For existing nodes *check for better path* (like Dijkstra's algorithm)



Algorithm A*

This variation courtesy S Baskaran

A*(S)

```
1  default value of g for every node is +∞
2  parent(S) ← null
3  g(S) ← 0
4  f(S) ← g(S) + h(S)
5  OPEN ← S : [ ]
6  CLOSED ← empty list
7  while OPEN is not empty
8      N ← remove node with lowest f value from OPEN
9      add N to CLOSED
10     if GoalTest(N) = true
11         return ReconstructPath(N)
12     for each neighbour M ∈ MoveGen(N)
13         if ( g(N) + k(N, M) ) < g(M)
14             parent(M) ← N
15             g(M) ← g(N) + k(N, M)
16             f(M) ← g(M) + h(M)
17             if M ∈ OPEN then continue
18             if M ∈ CLOSED then Propagate-Improvement(M)
19             else add M to OPEN        ▷ M is new
20 return empty list
```

In the style of Dijkstra's algorithm

Priority queue

Better path found?

Case 2: Neighbour in OPEN

Case 3: Neighbour in CLOSED

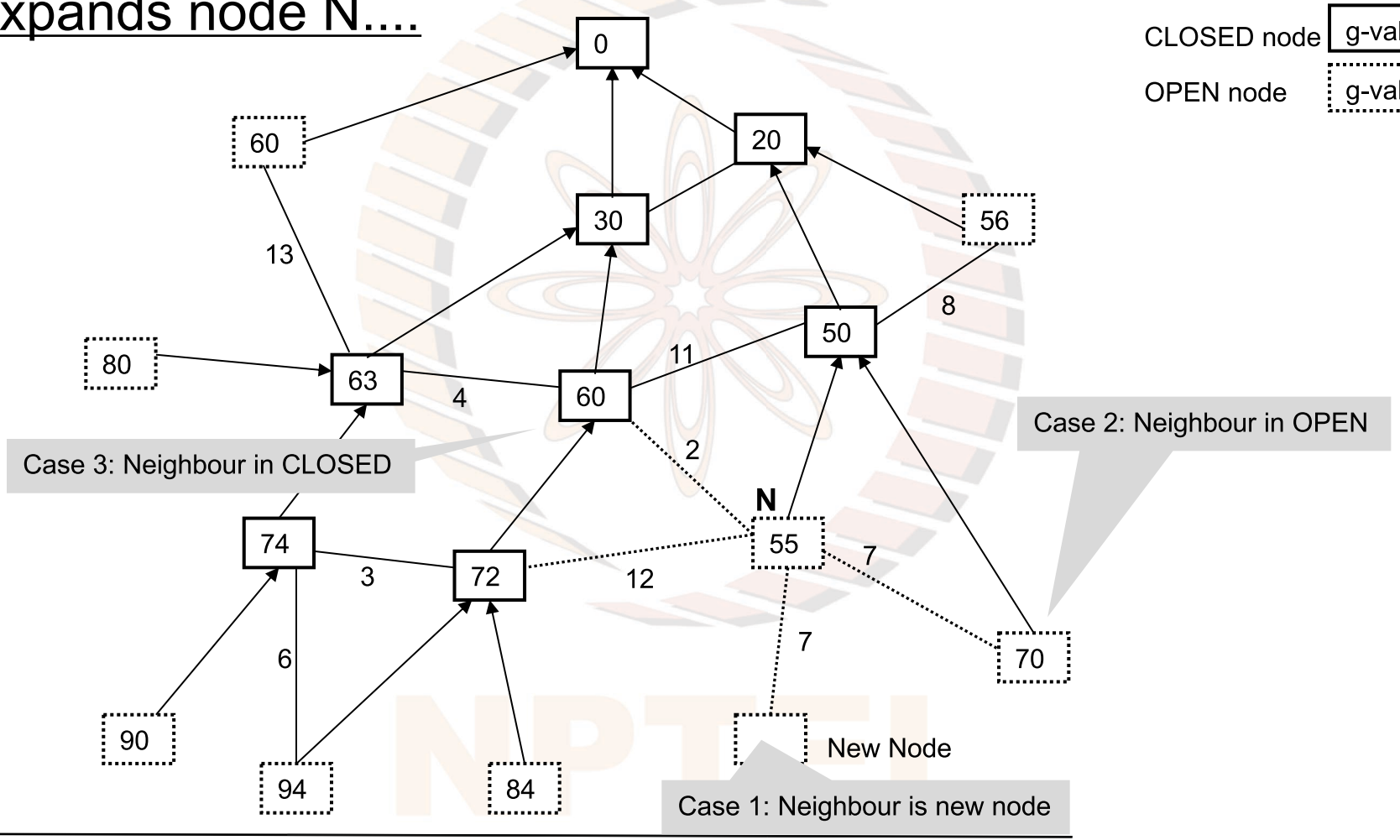
Case 1: Neighbour is new node

Algorithm A*: Case 3 – if better path to node on CLOSED found

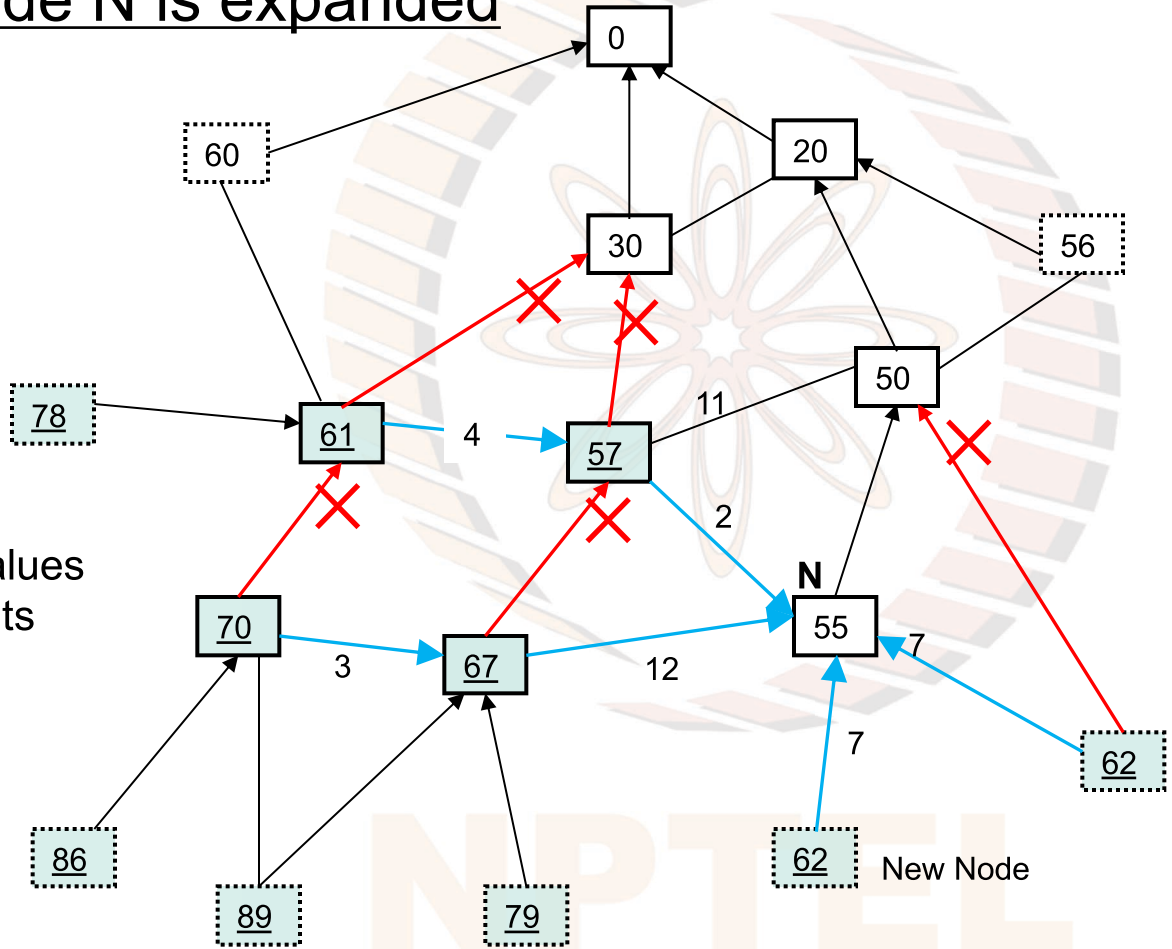
This variation courtesy
S Baskaran

```
PROPAGATE-IMPROVEMENT(M)
1  for each neighbour X ∈ MOVEGEN(M)
2      if ( g(M) + k(M, X) ) < g(X)        Better path found to X?
3          parent(X) ← M                   Update
4          g(X) ← g(M) + k(M, X)
5          f(X) ← g(X) + h(X)
6          if X ∈ CLOSED                   Recurse
7              PROPAGATE-IMPROVEMENT(X)
```

A* expands node N....



After node N is expanded



CLOSED node g-value

OPEN node g-value

Lower g-values
New parents

Next

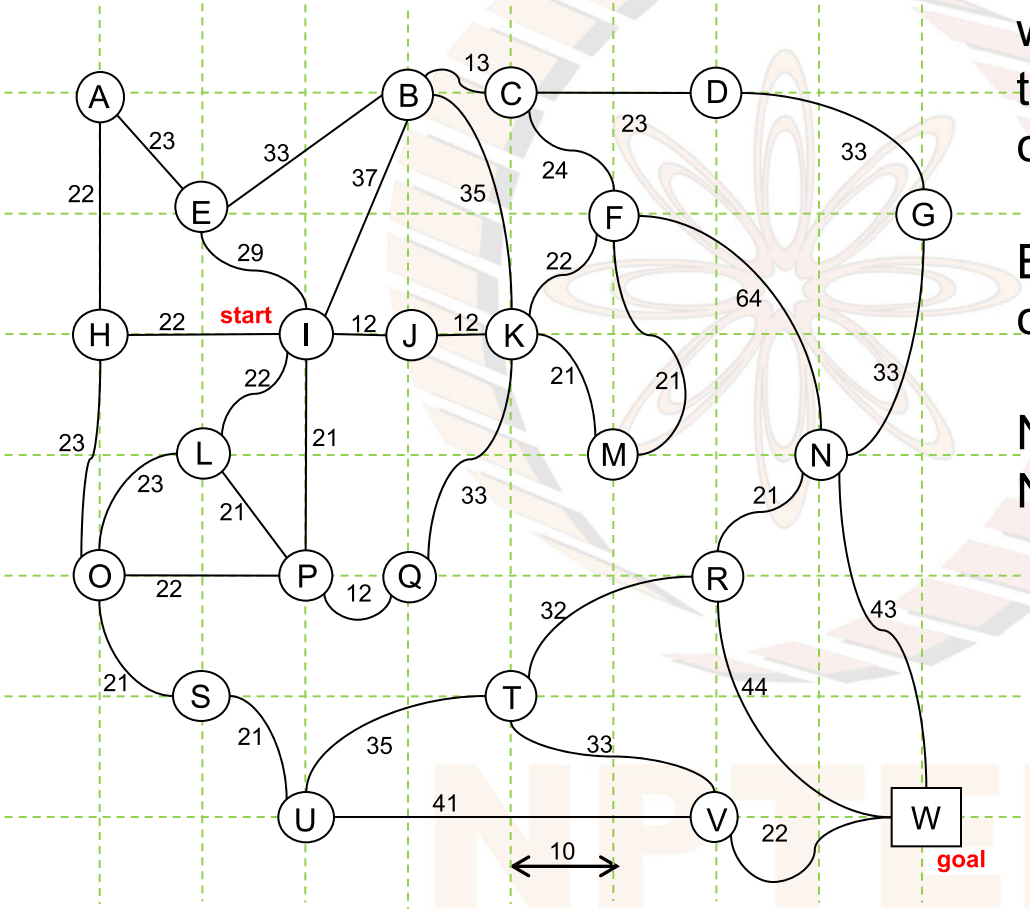
An illustration of how A* searches the implicit graph,

guided by the f-values,

generating the nodes on the fly,

till it picks a goal node.

A problem graph



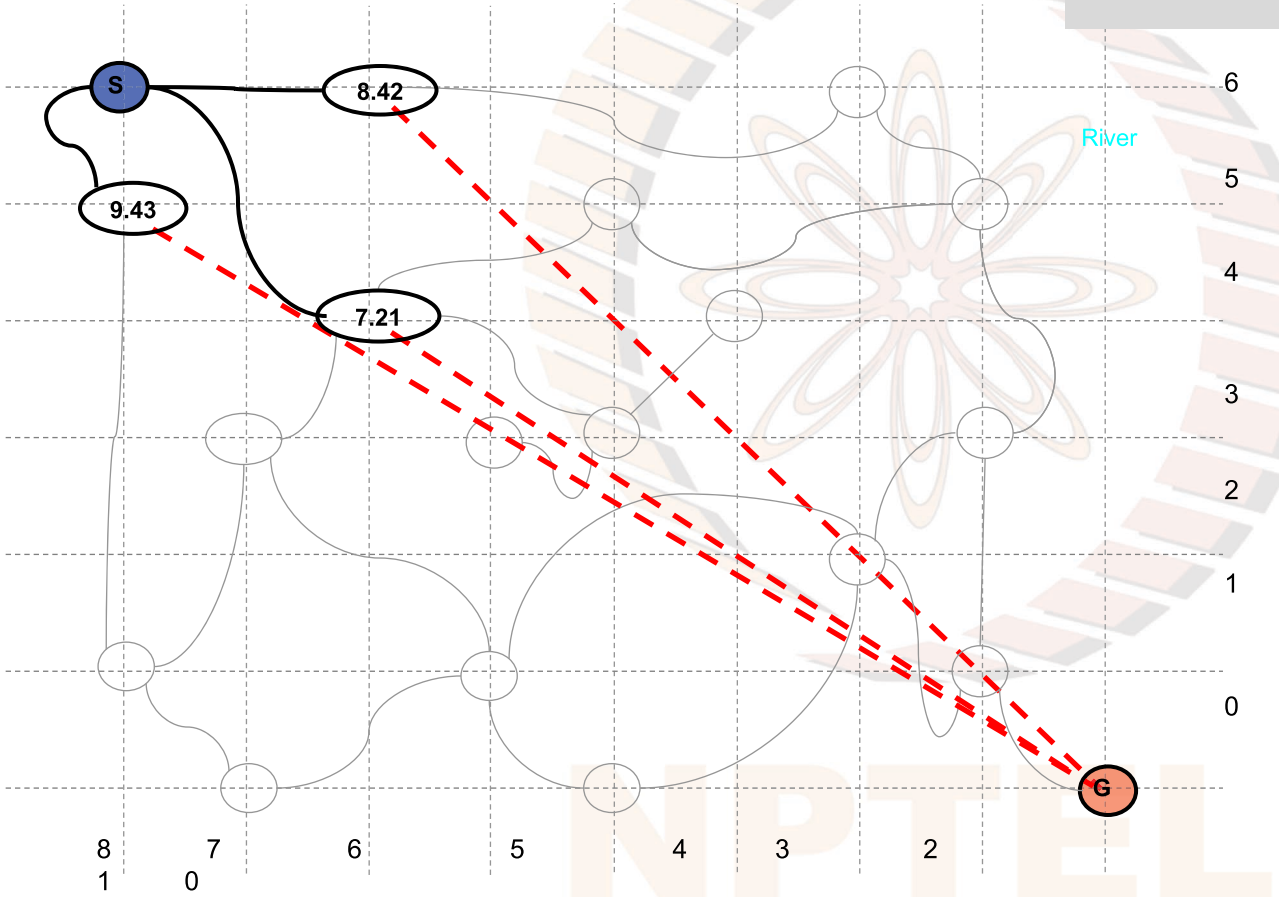
The nodes are placed on a grid where each edge is 10km, and their coordinates can be computed easily

Each edge is labeled with the cost of traversing that edge

Node I is the start node
Node W is the goal node

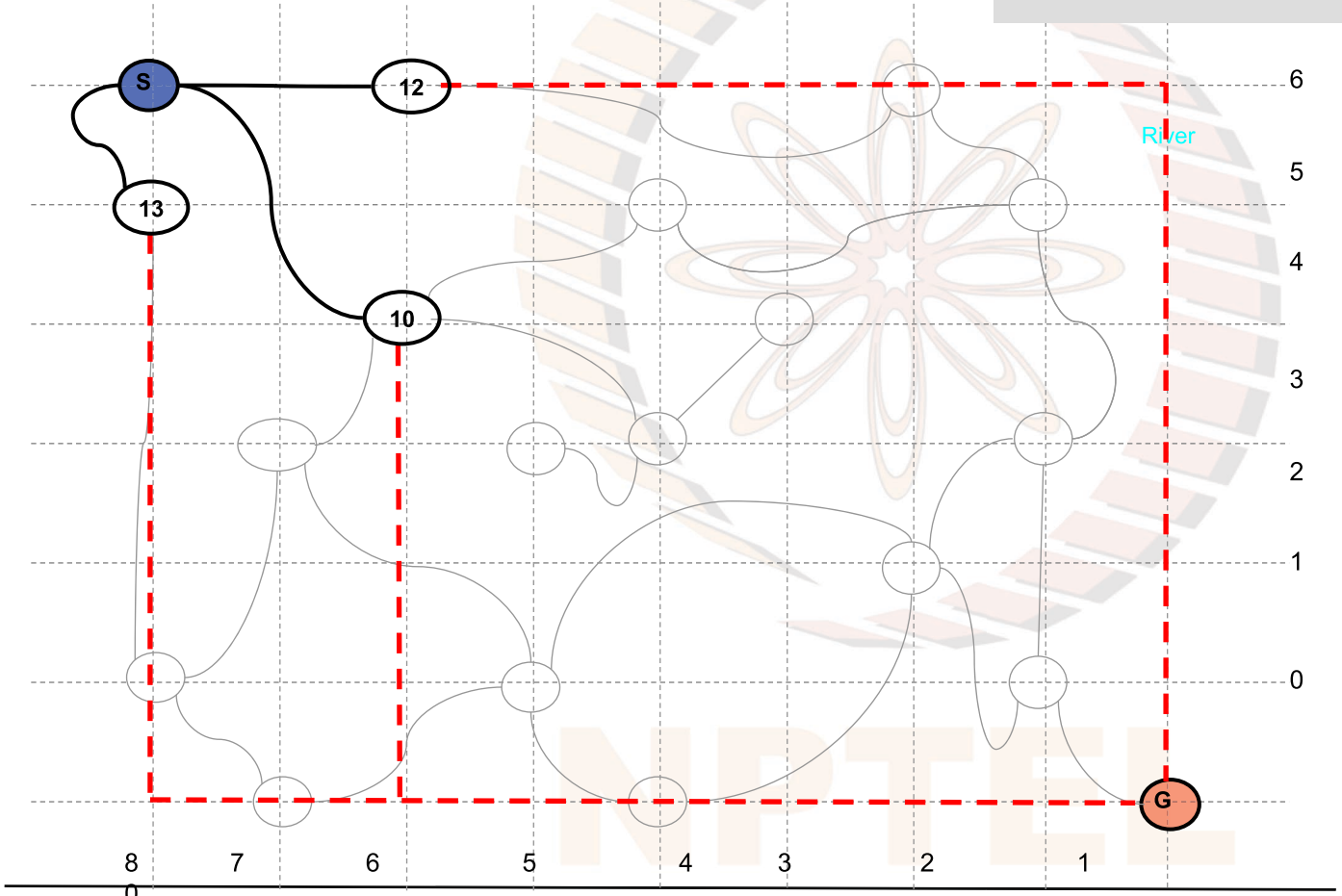
h(n) = Euclidean distance (recap)

Euclidean: h(n) = $\sqrt{(x_{Goal} - x_n)^2 + (y_{Goal} - y_n)^2}$



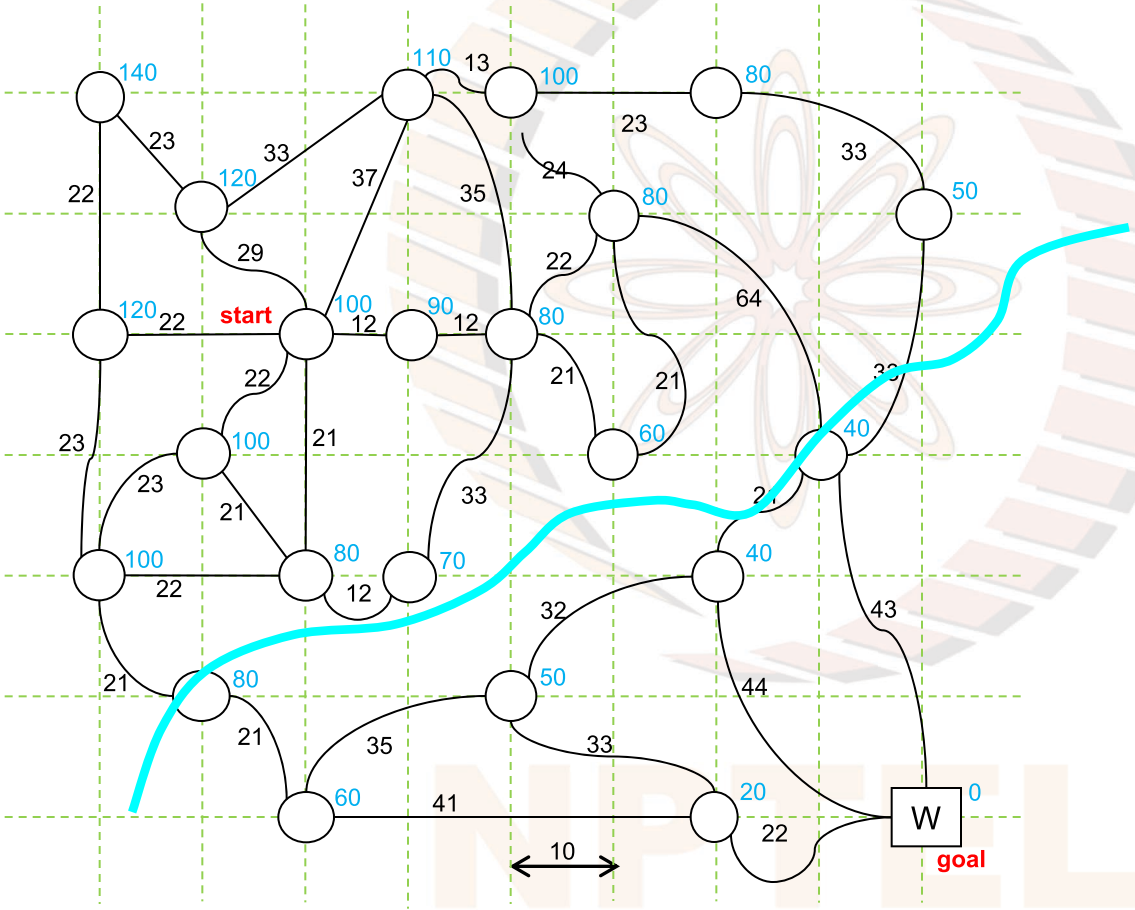
h(n) = Manhattan distance (recap)

Manhattan: $h(n) = |x_{Goal} - x_n| + |y_{Goal} - y_n|$

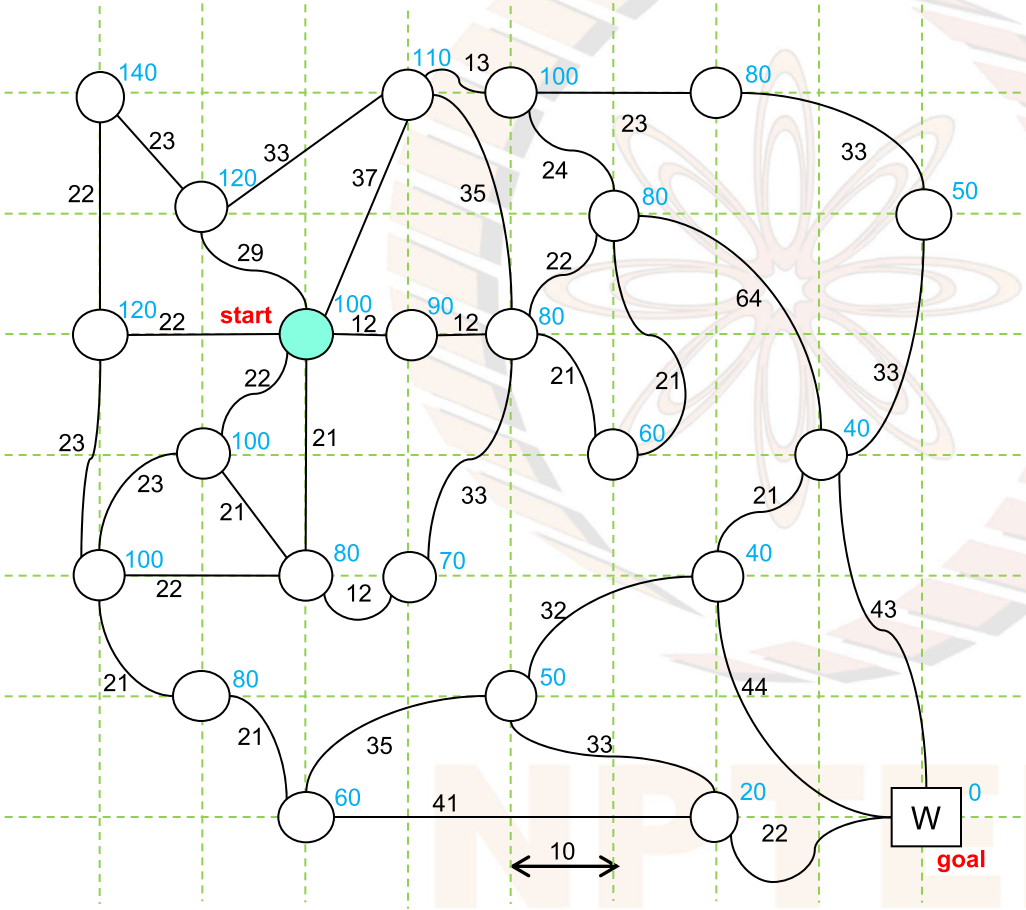


We will use the Manhattan distance for simplicity

The Manhattan distance values for each node



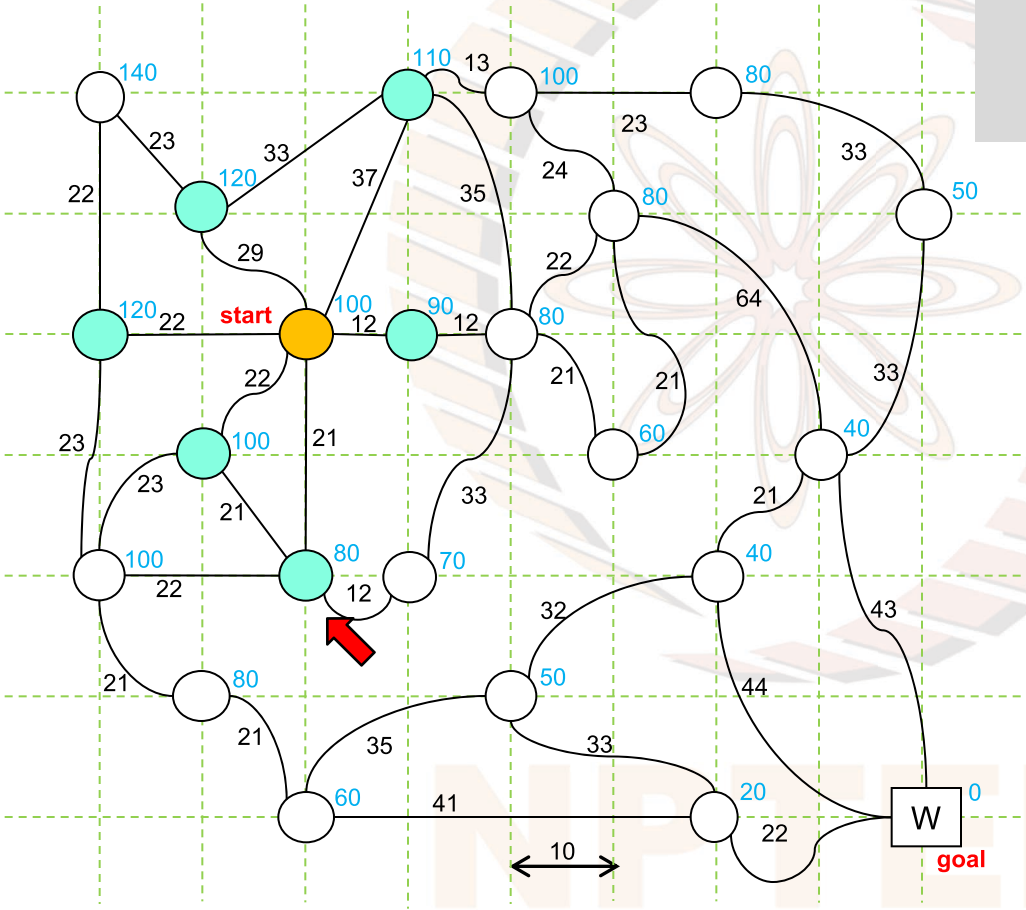
Best First Search



h(start) = 100

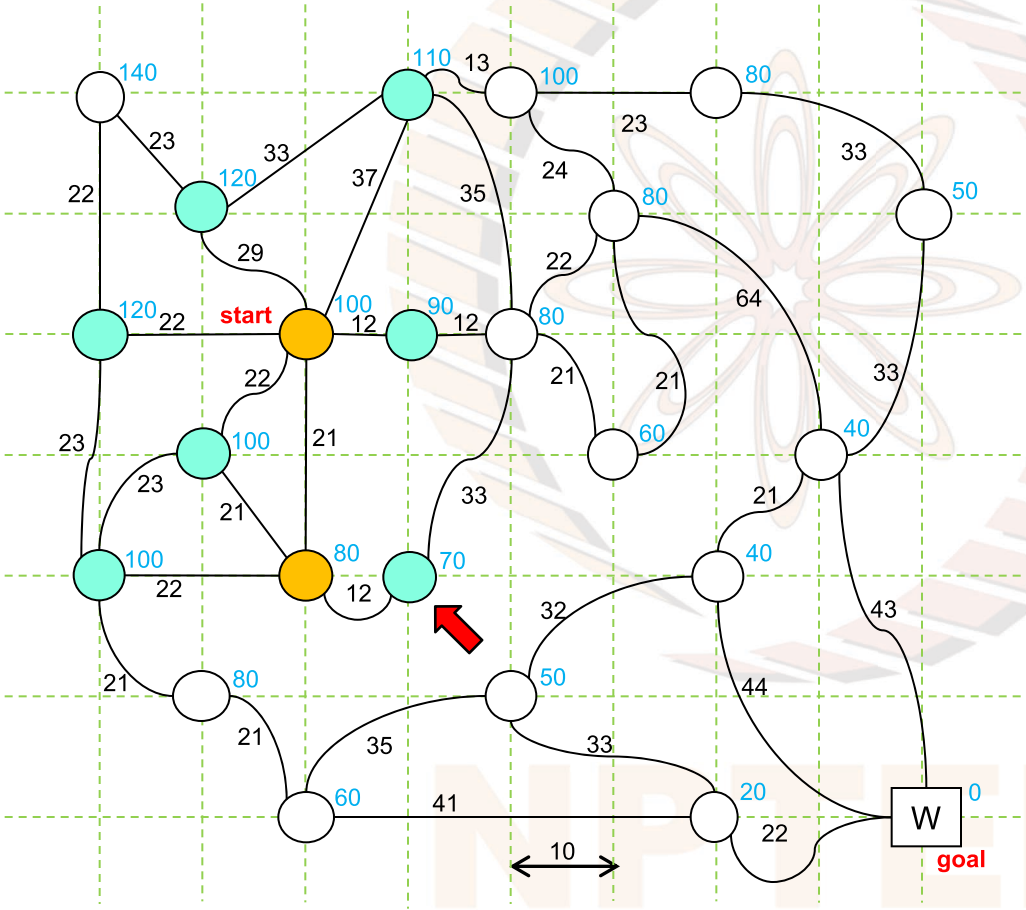
Node with lowest h-value selected at each stage from OPEN

Best First Search



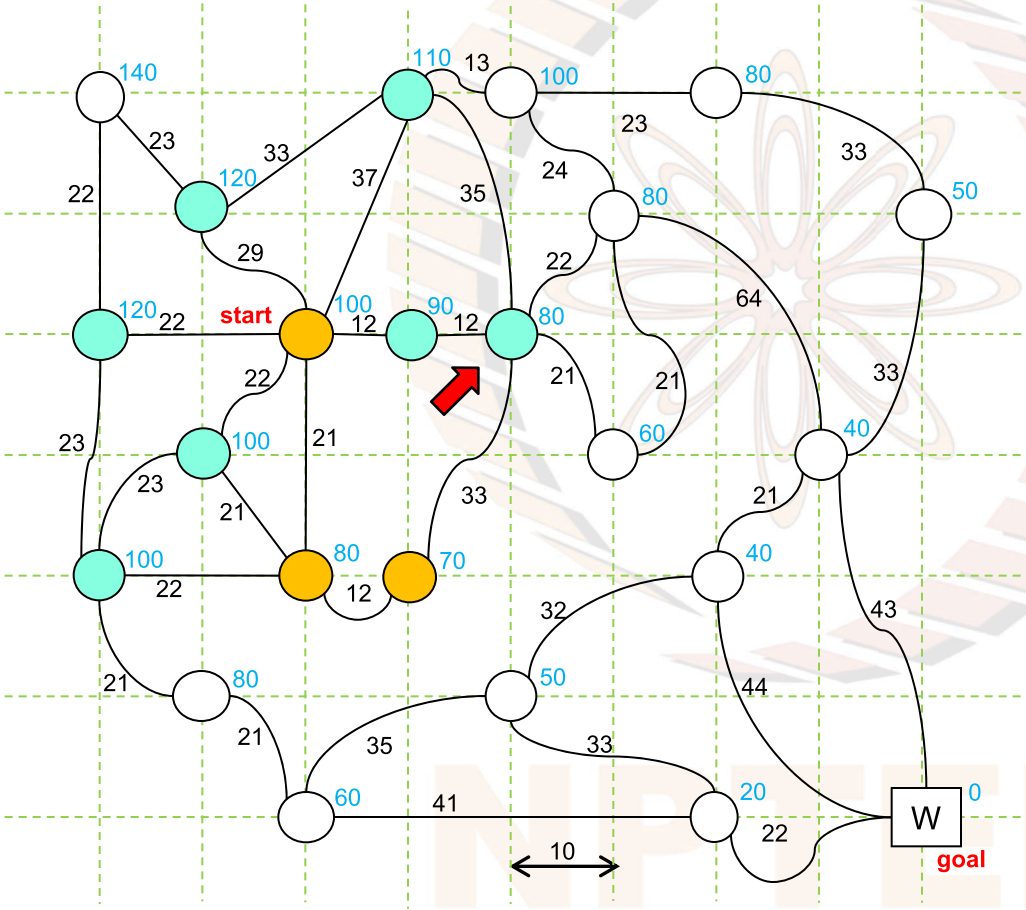
Arrow points to the best node on OPEN

Best First Search

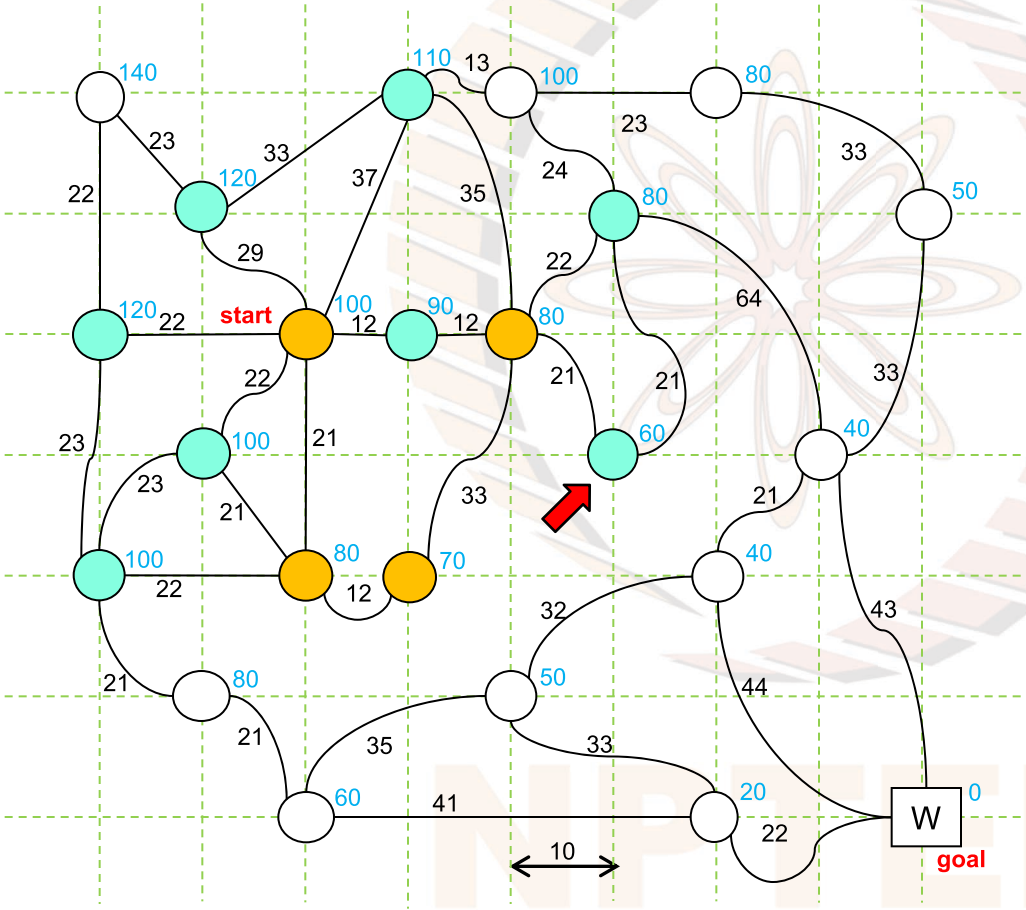


Hill Climbing would have got stuck here at h-value 70.

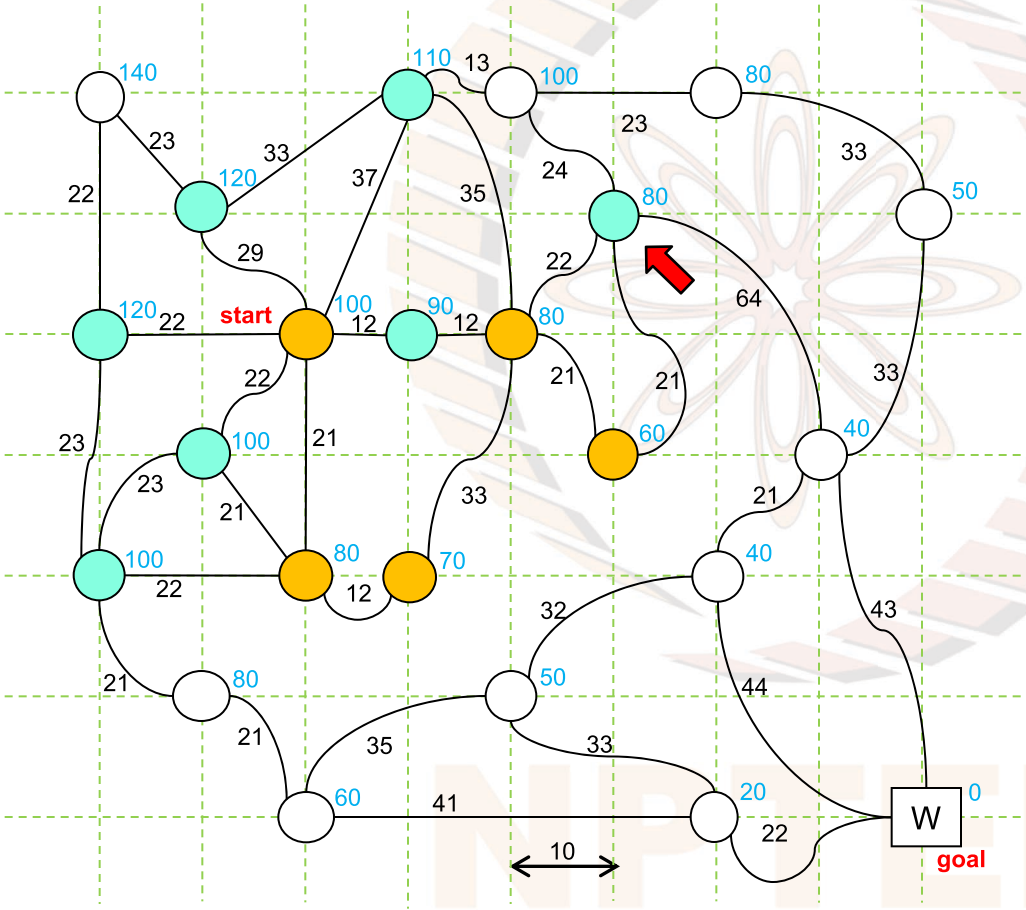
Best First Search



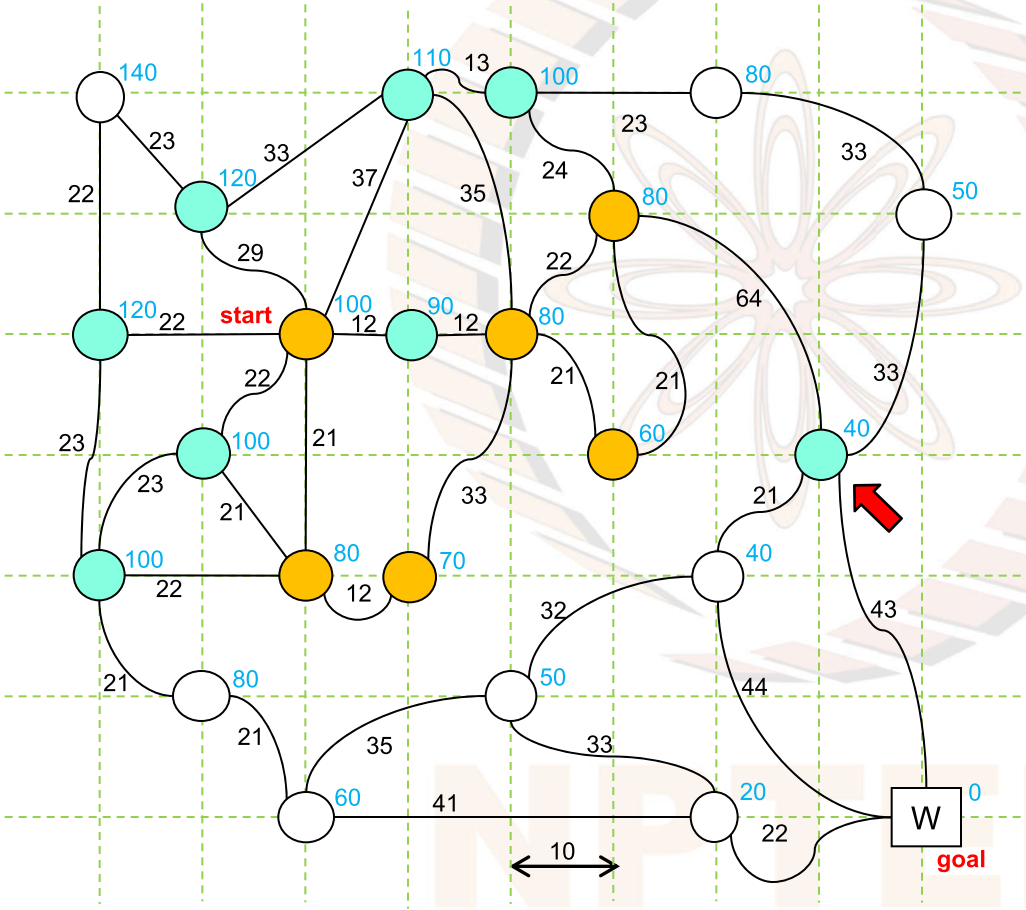
Best First Search



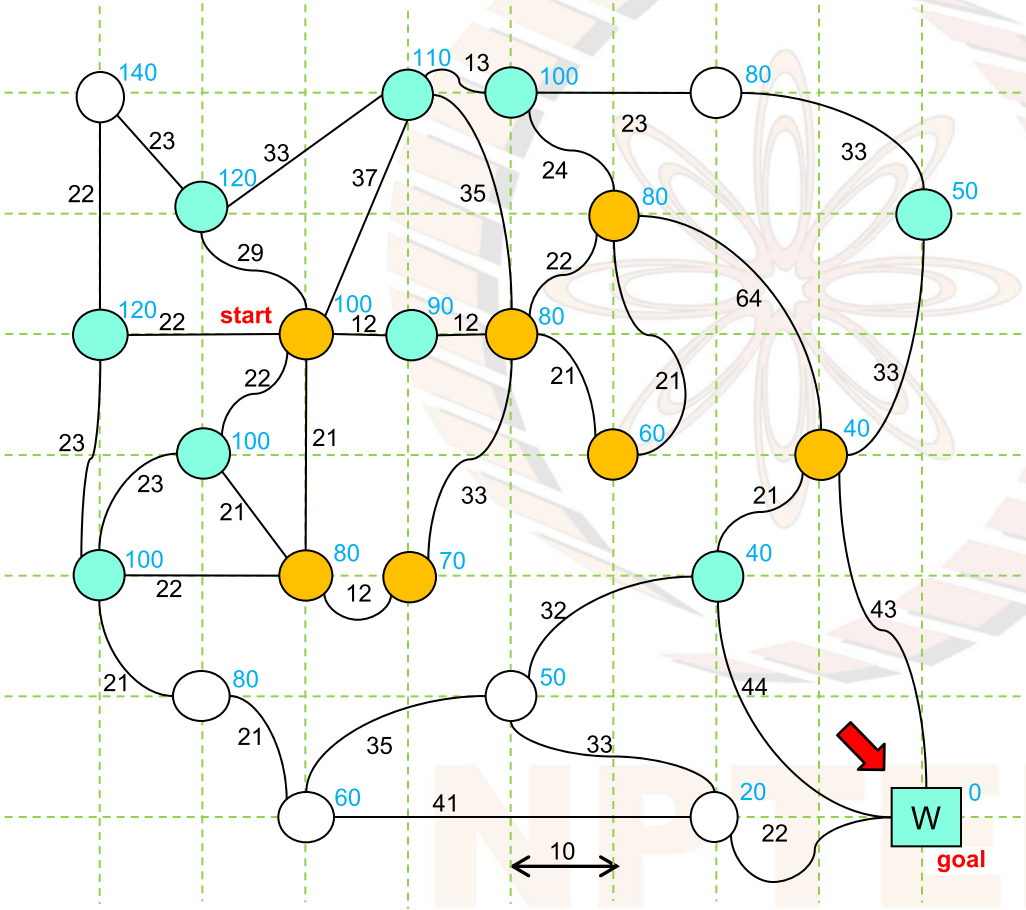
Best First Search



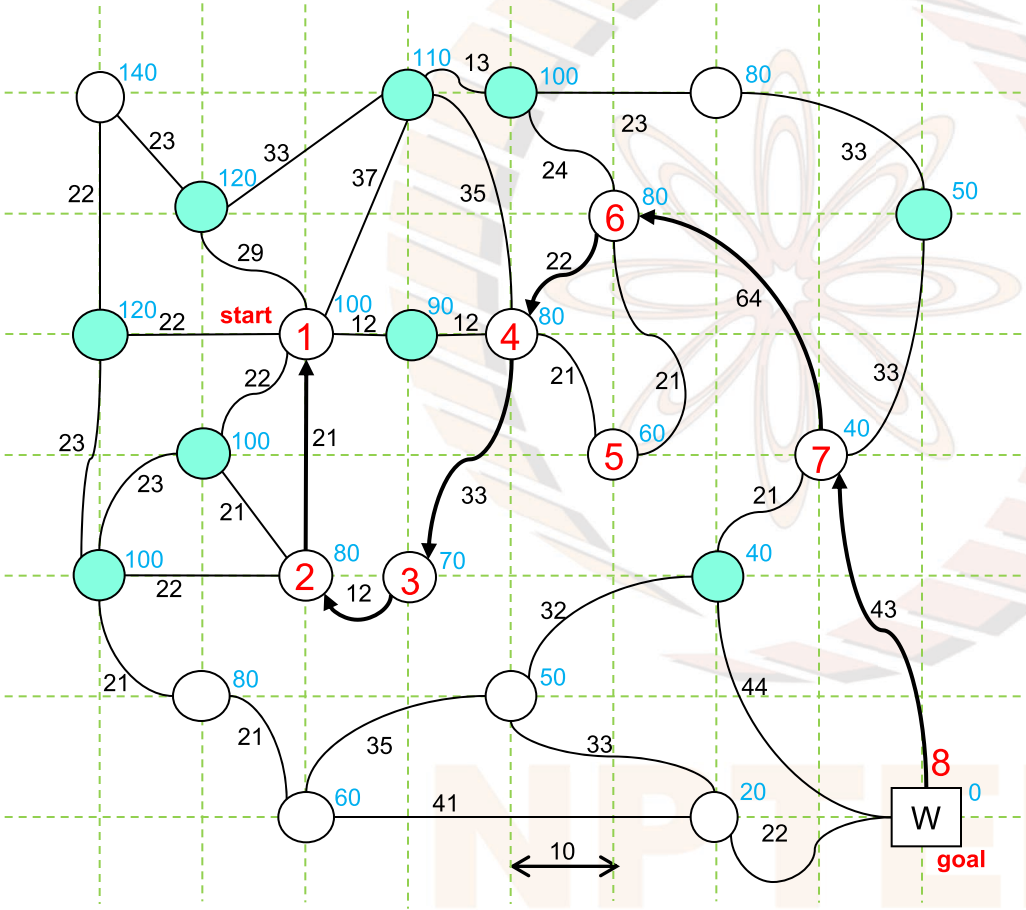
Best First Search



Best First Search

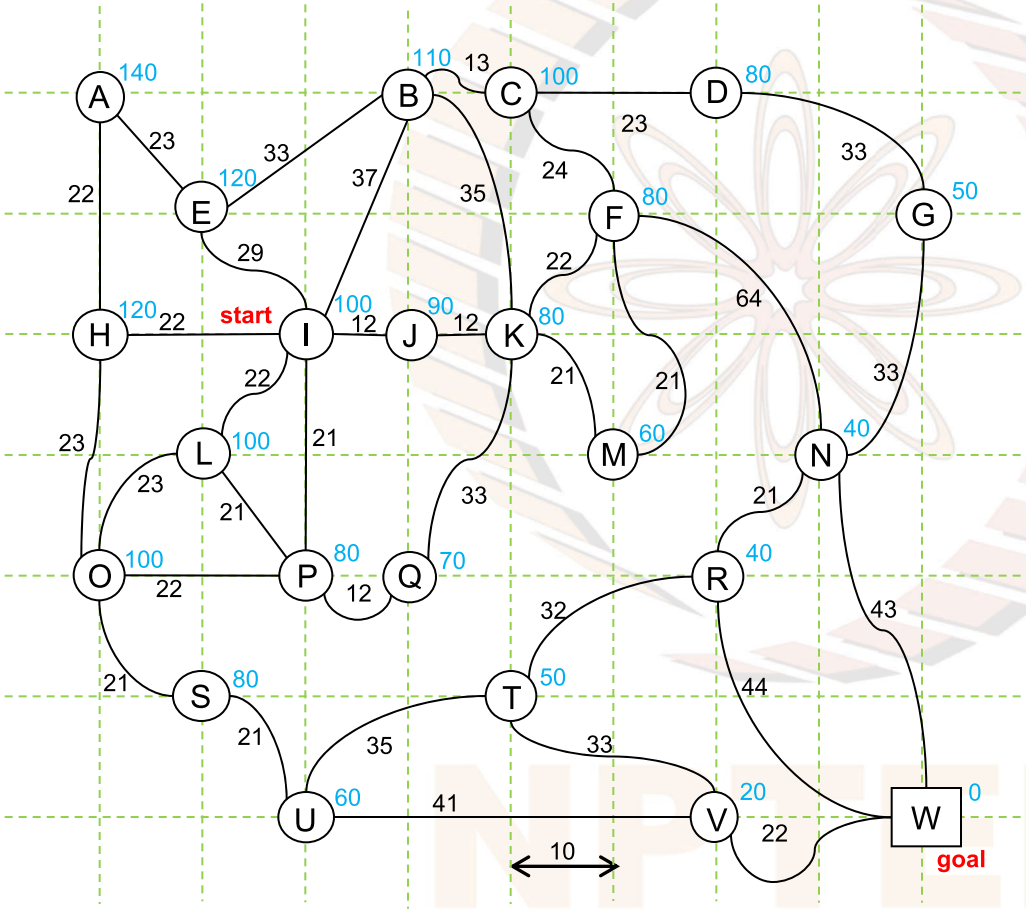


Path found by Best First Search

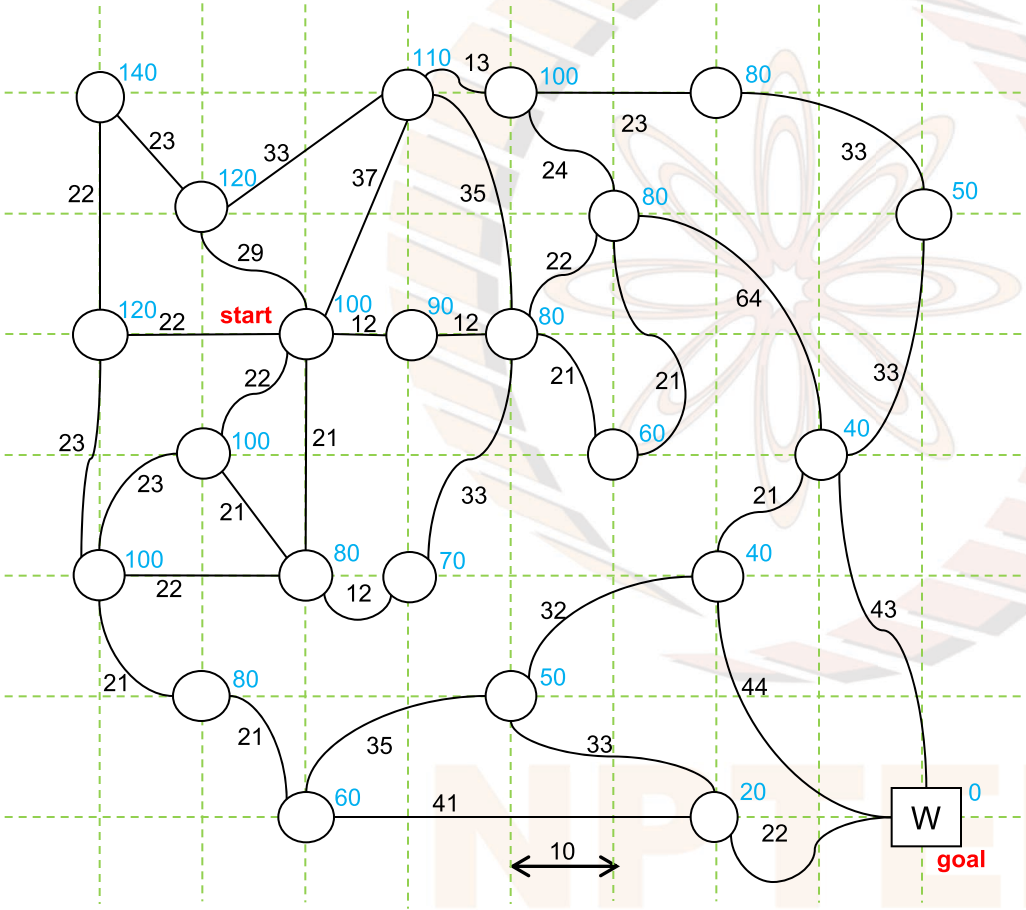


8 nodes inspected
Cost of Solution = 195

A*: An example


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
start
goal
140
110
100
80
120
80
50
120
100
90
80
100
60
40
100
80
70
40
80
50
60
20
0
23
33
13
23
24
33
22
37
35
29
22
64
22
12
12
21
21
33
23
21
23
21
33
21
22
12
32
43
21
44
21
35
33
41
22
10

A* example: label nodes with their f-values

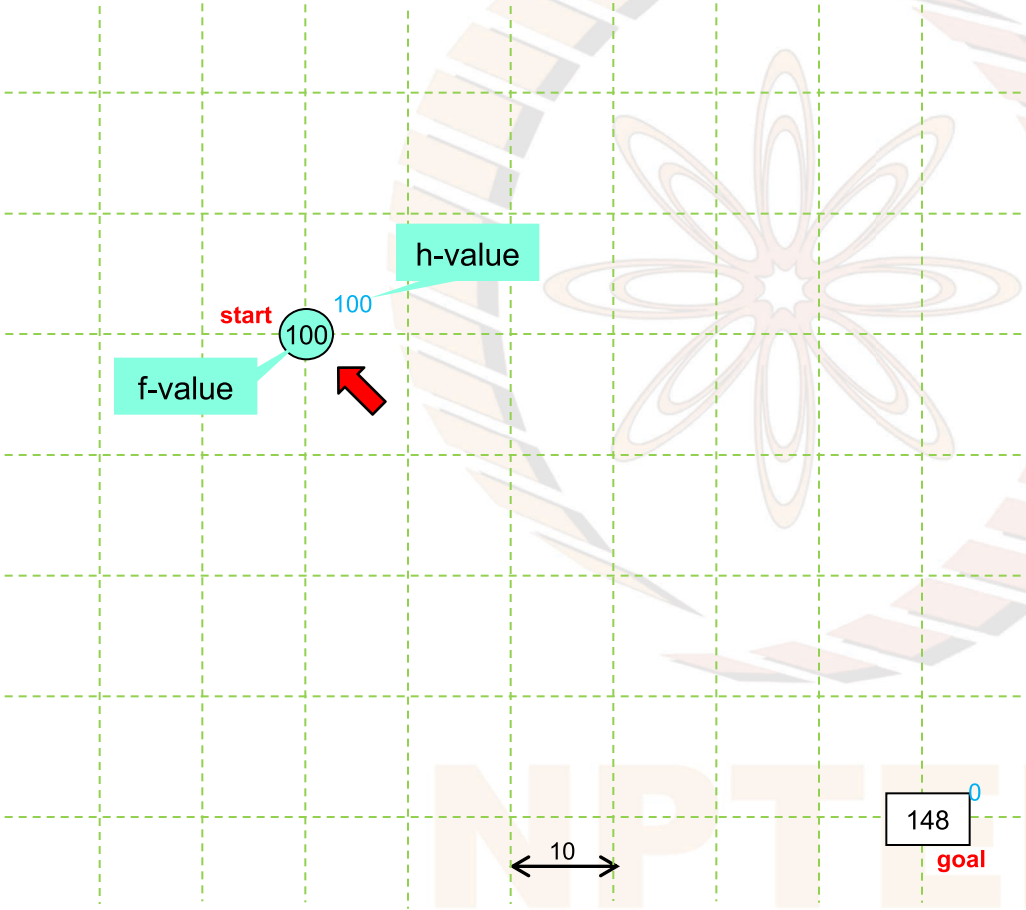


For the start node
f(start) = h(start)

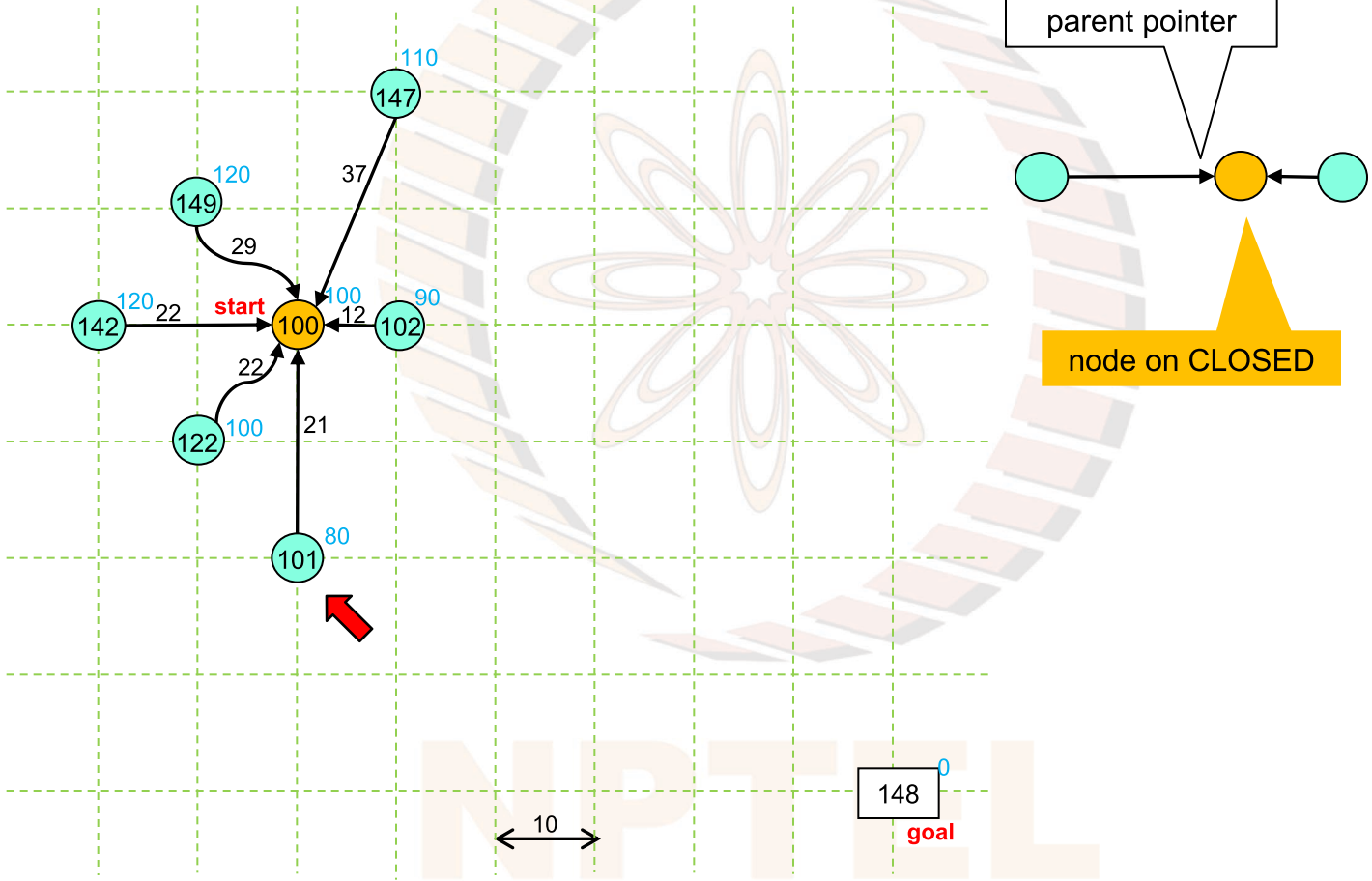
Initialize OPEN with the start node

A* starts with the start node in OPEN

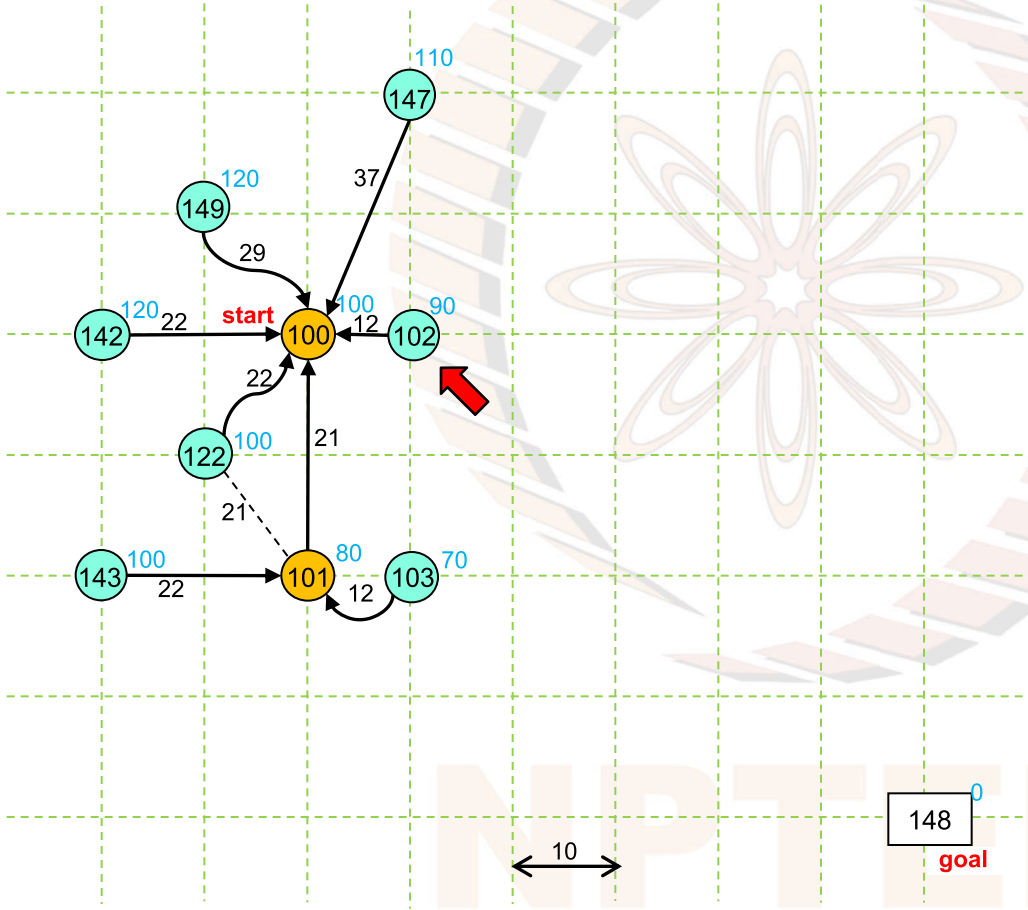
A* too begins by inspecting the start node



A*: The Start node has 6 children

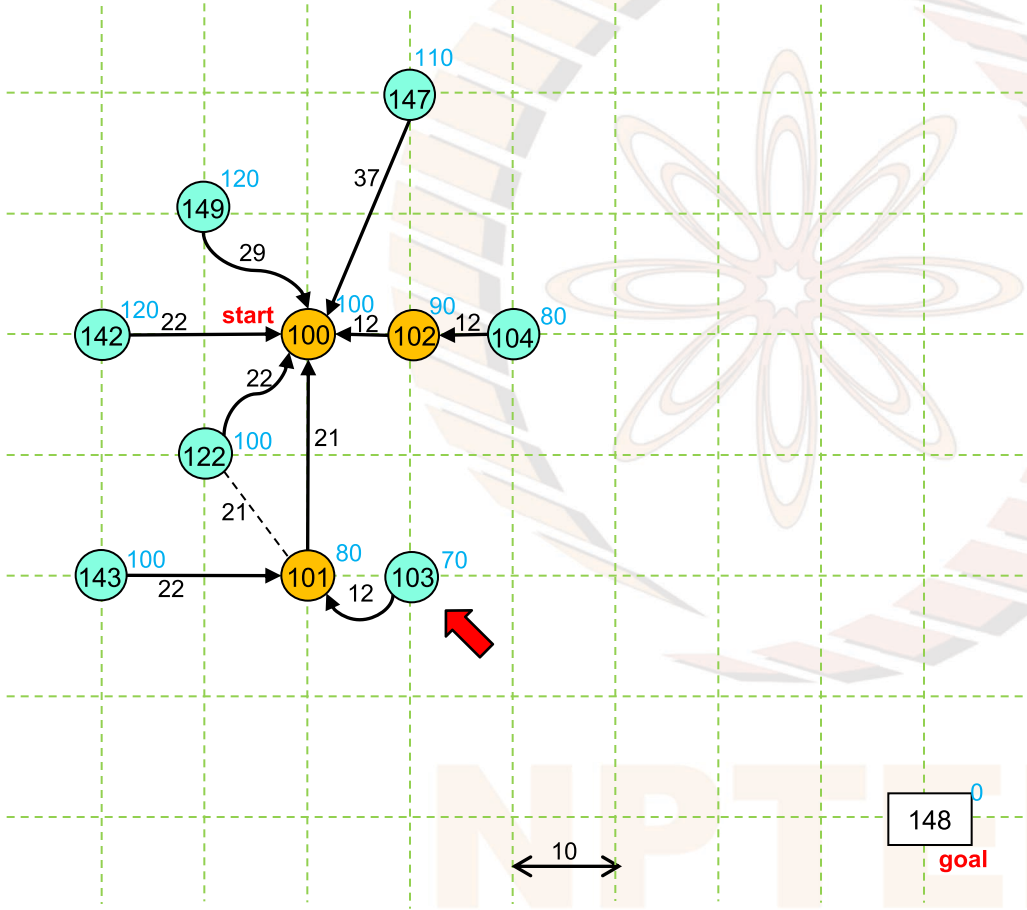


A*: Inspect the best node in OPEN

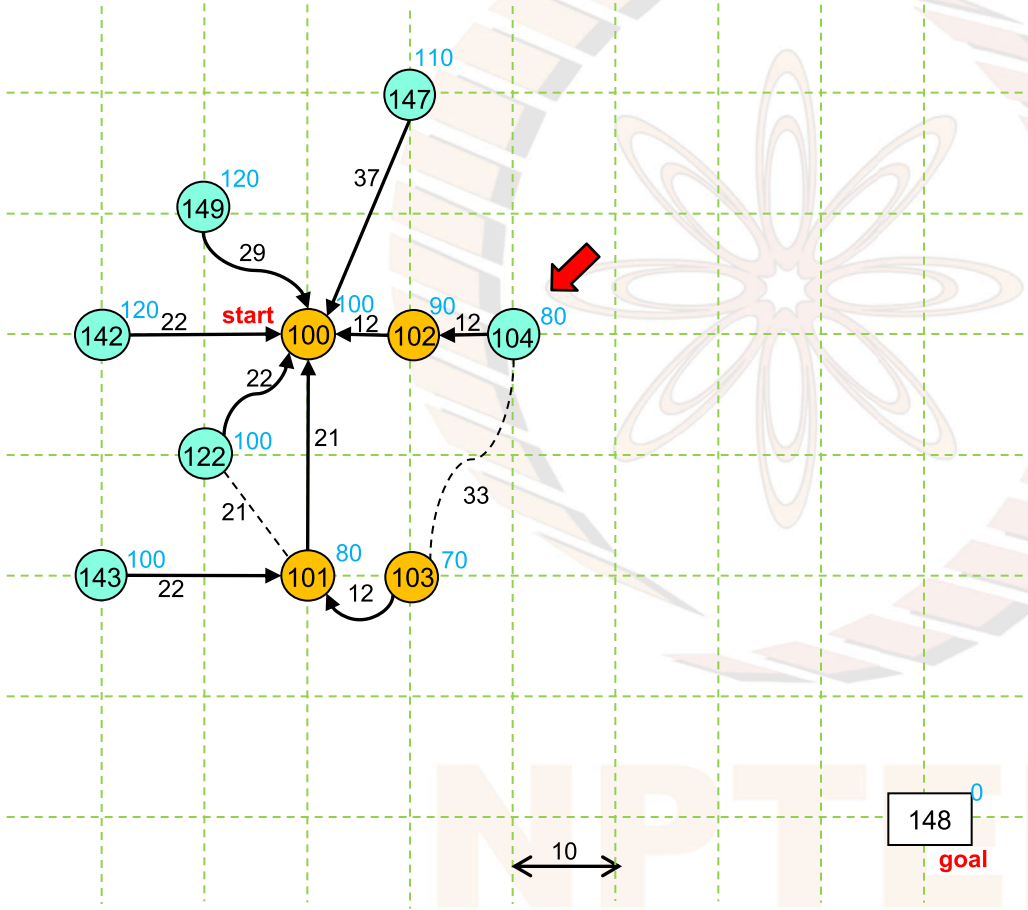


A* does pick this node ignored by Best First

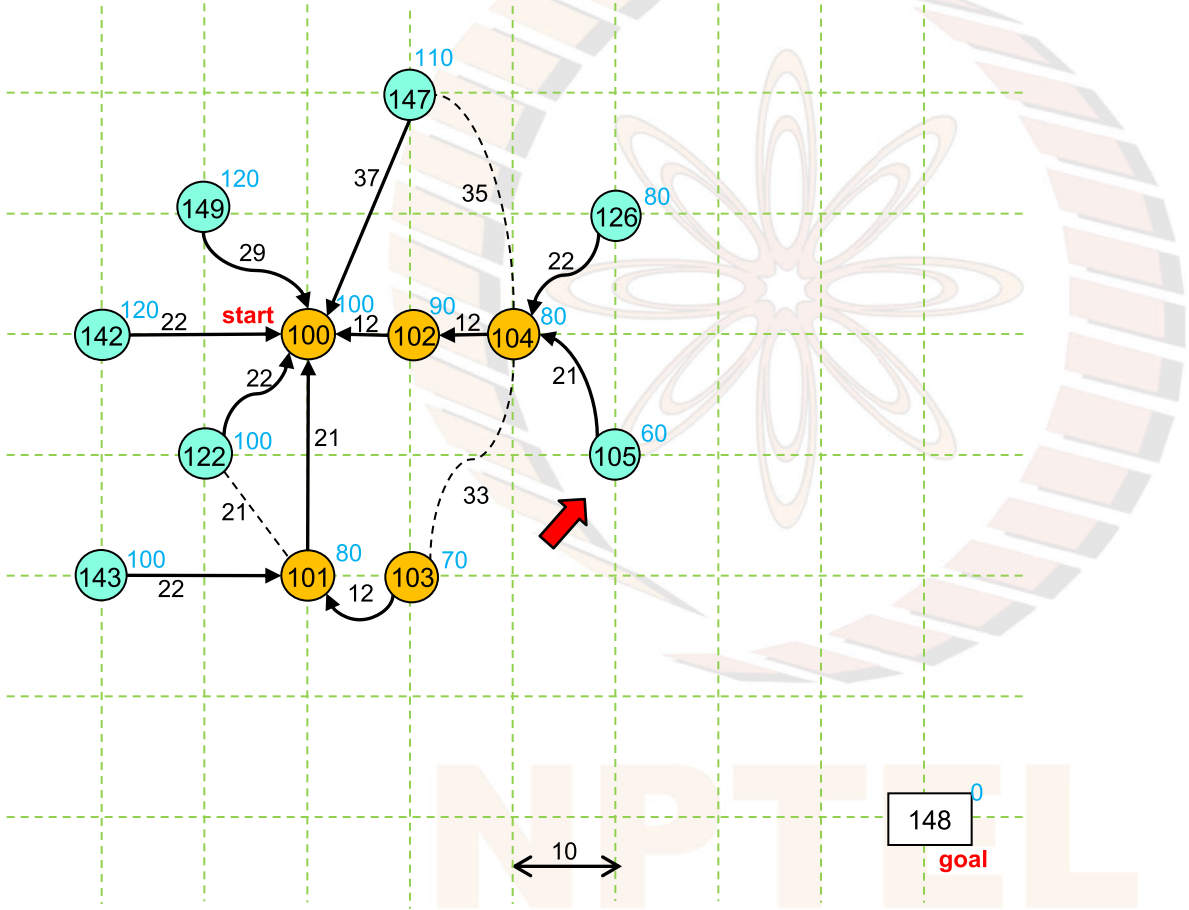
A*: Inspect the best node in OPEN



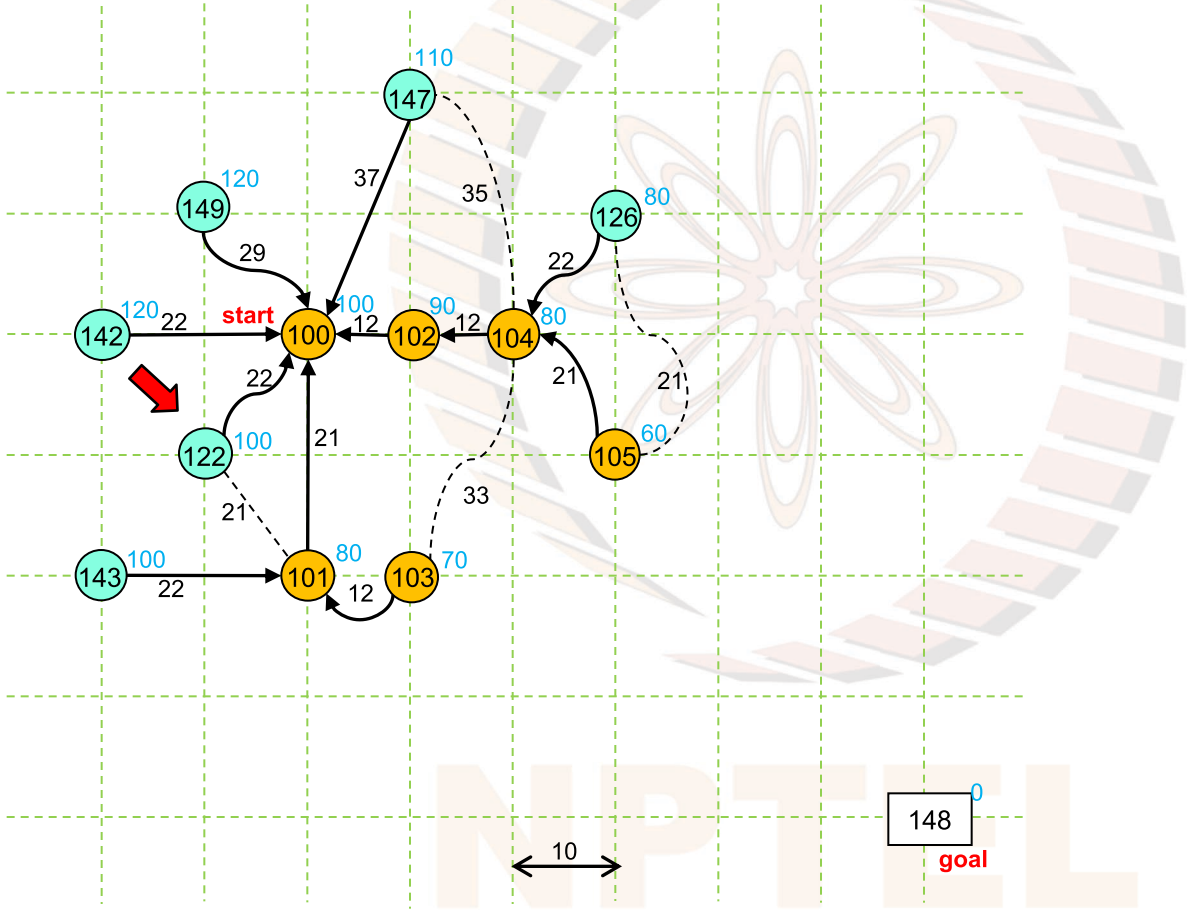
A*: Inspect the best node in OPEN



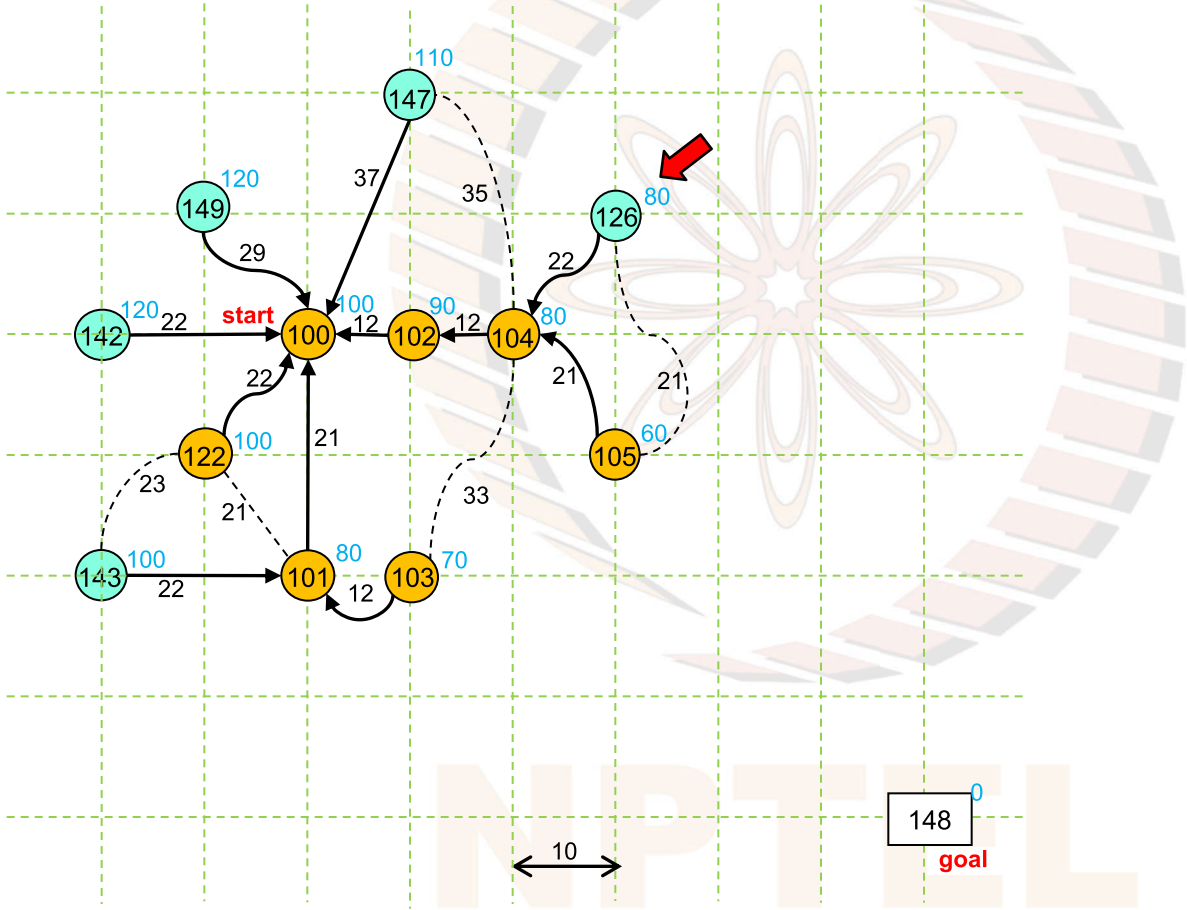
A*: Inspect the best node in OPEN



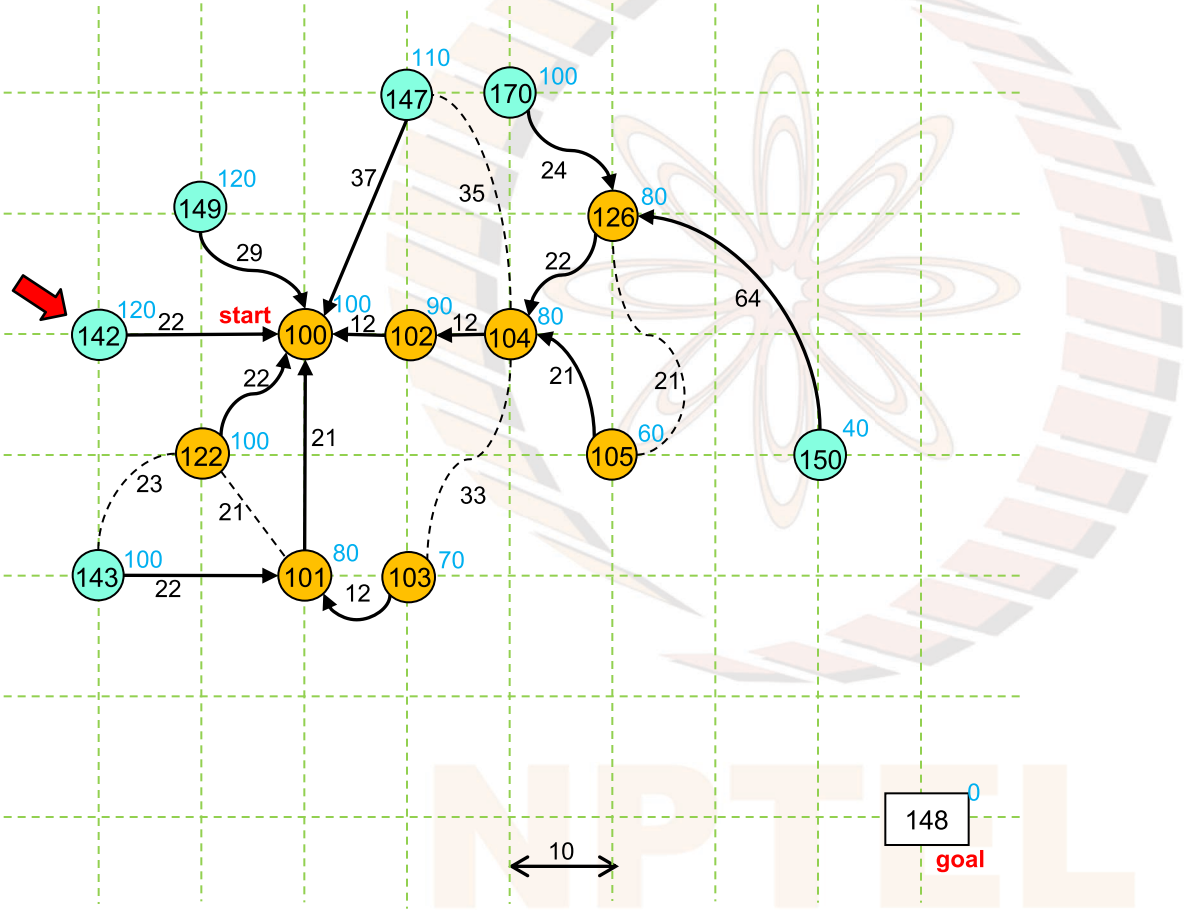
A*: Inspect the best node in OPEN



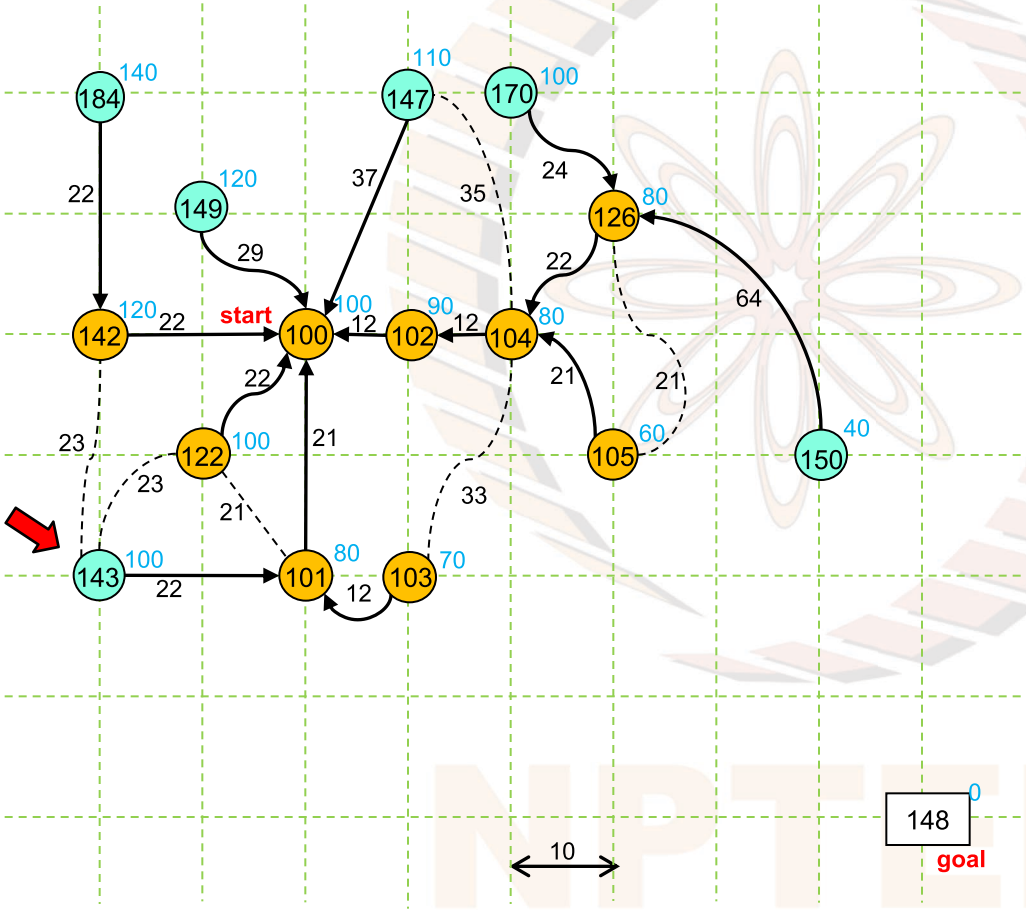
A*: Inspect the best node in OPEN



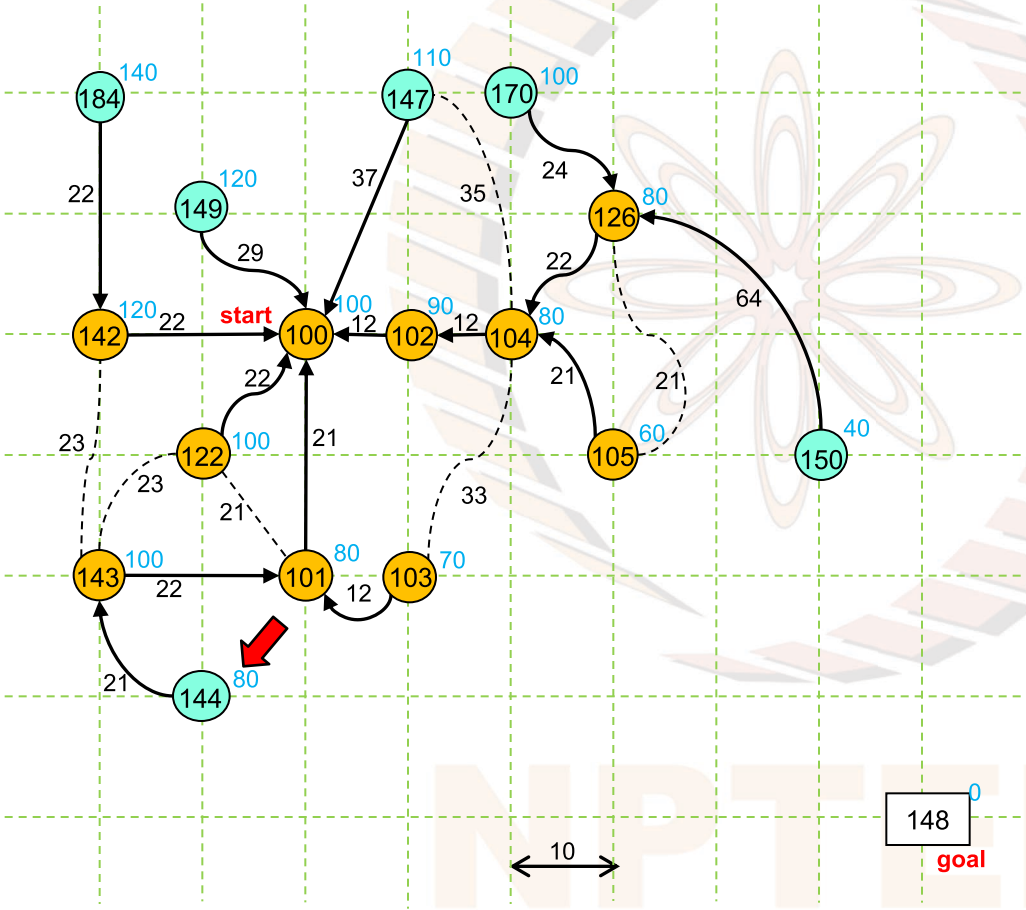
A*: Inspect the best node in OPEN



A*: Inspect the best node in OPEN



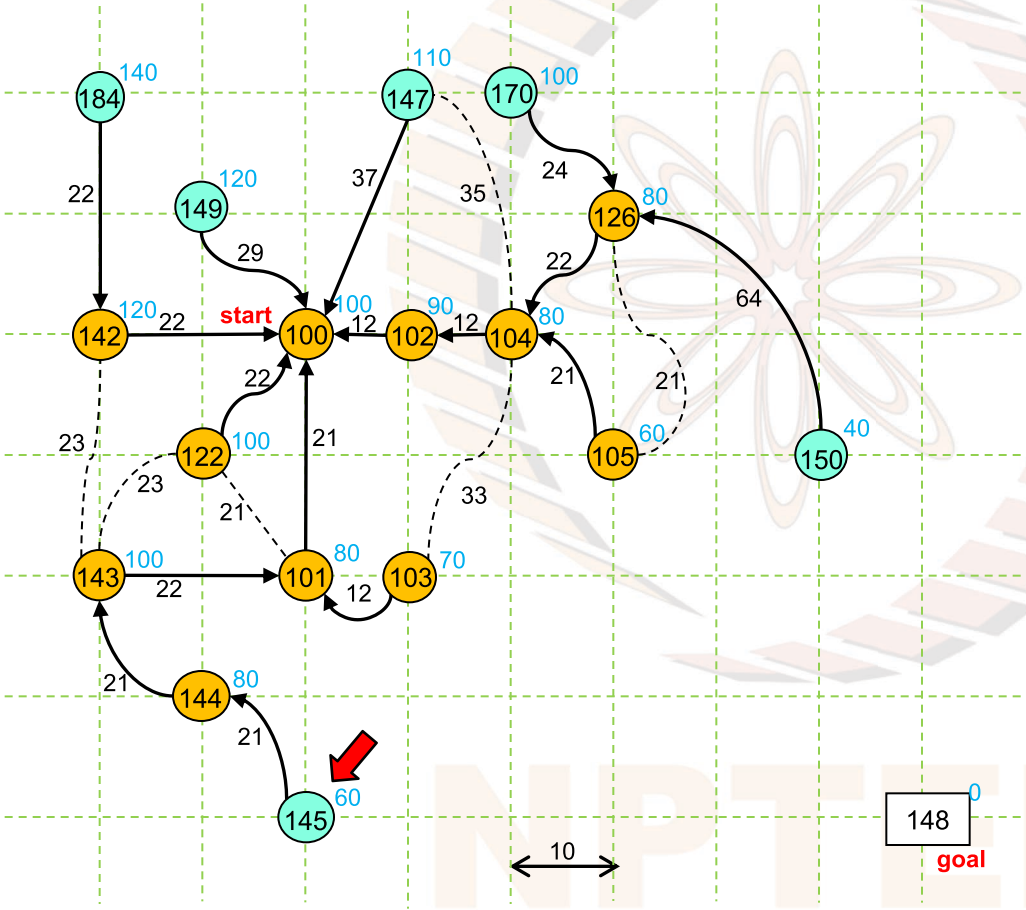
A*: Inspect the best node in OPEN



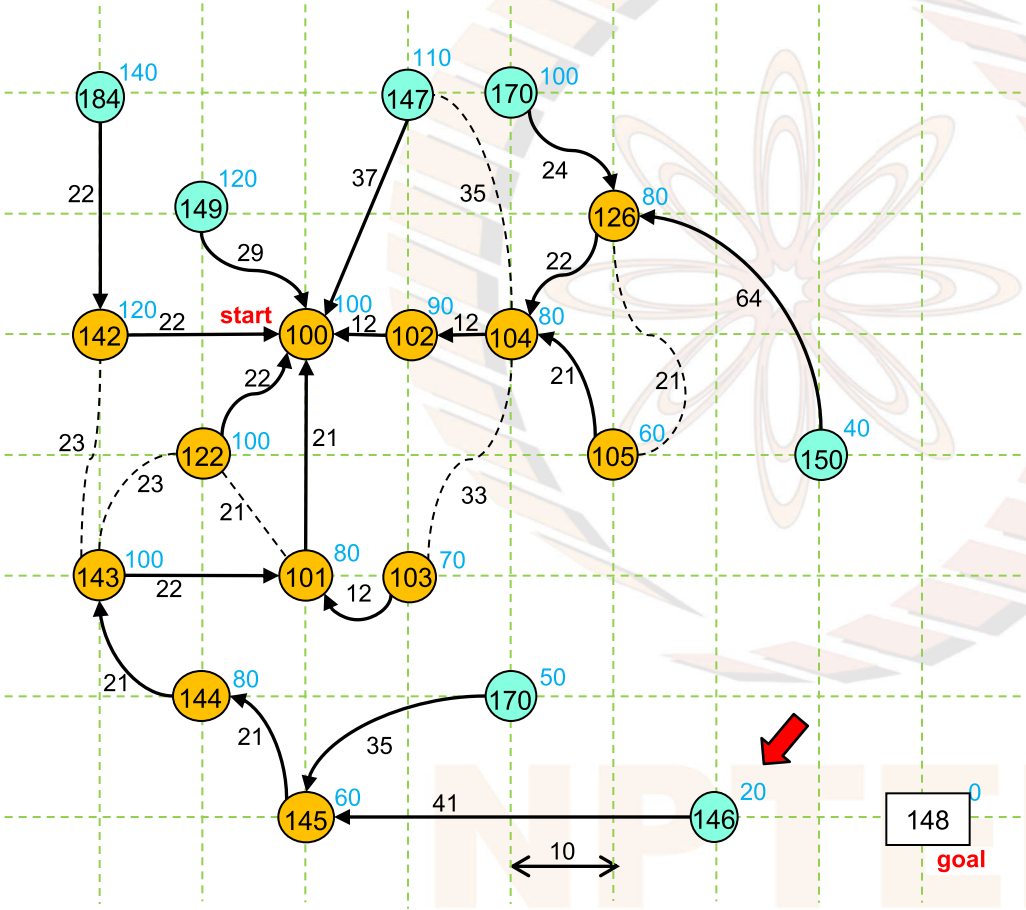
A* is *much more* explorative than Best First

Needed for searching for optimal path!

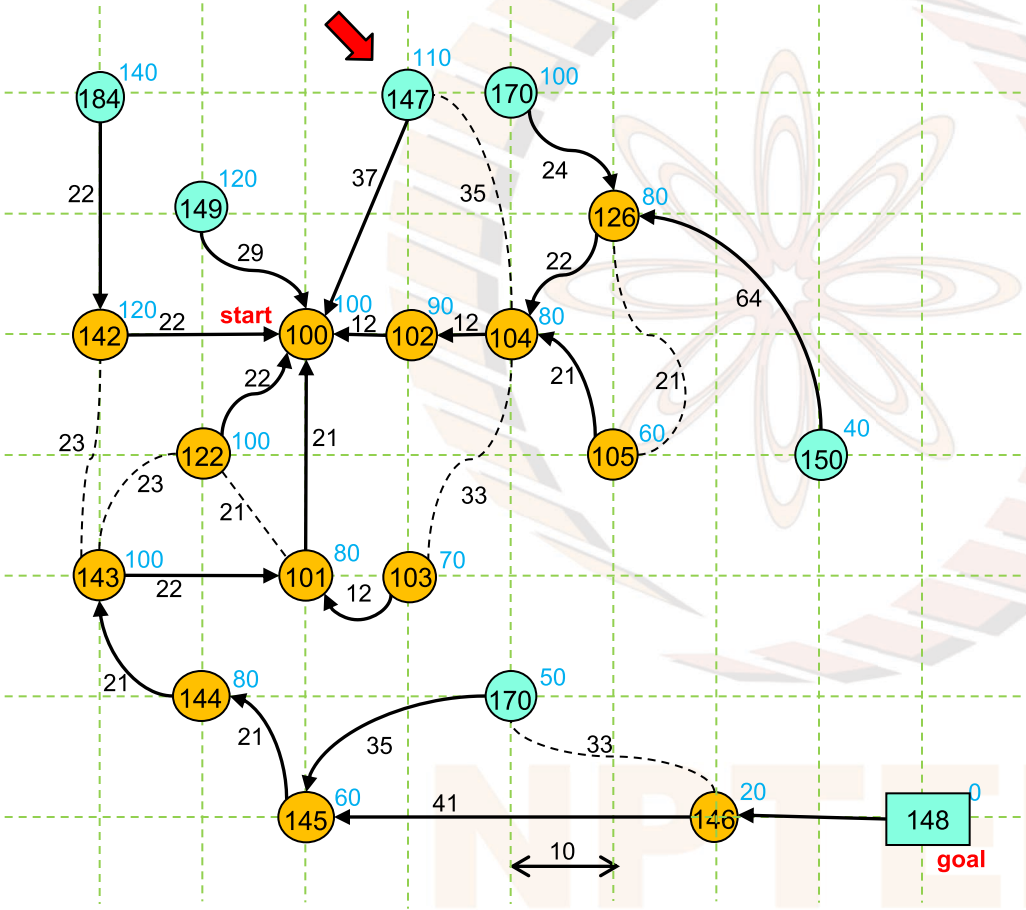
A*: Inspect the best node in OPEN



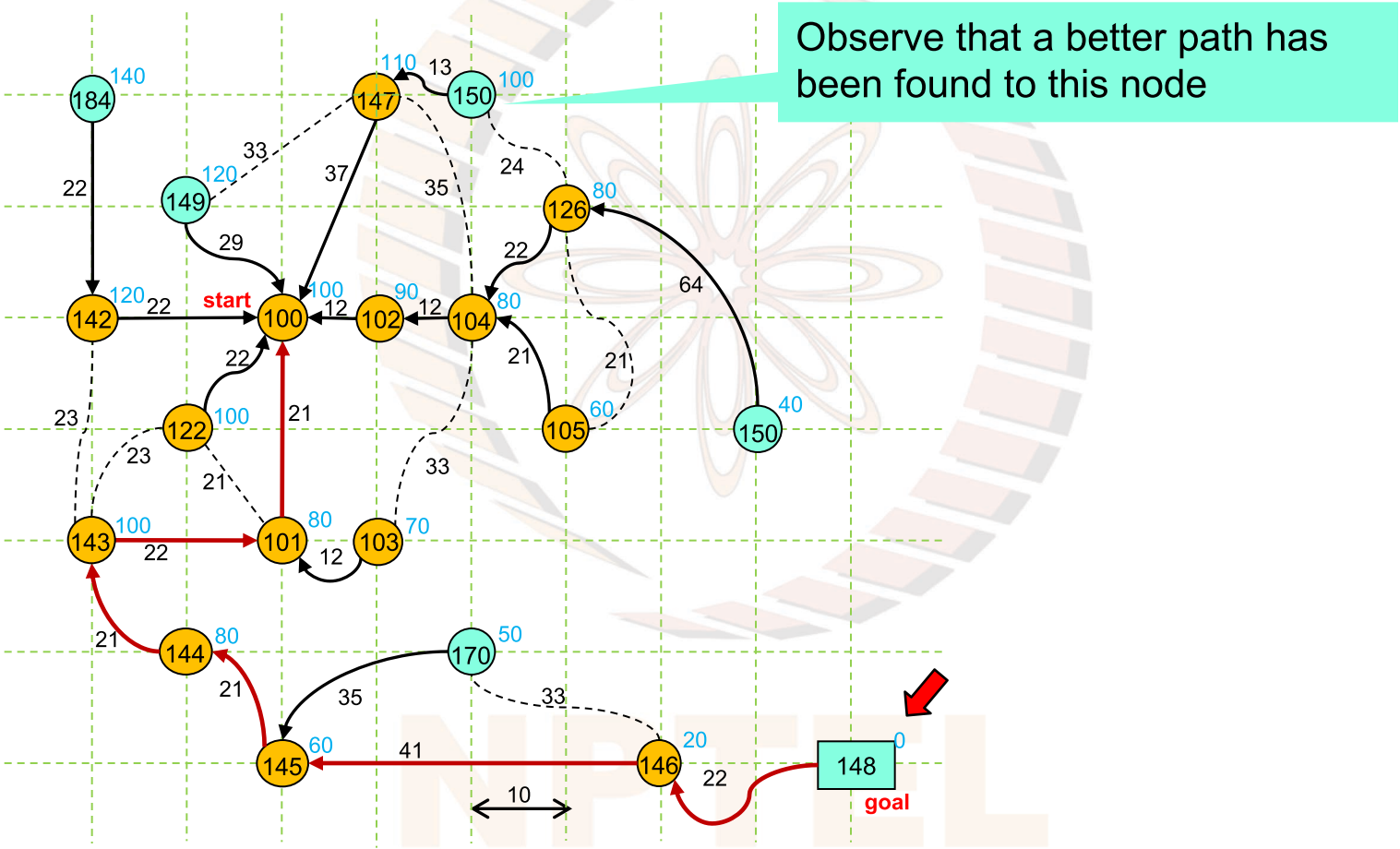
A*: Inspect the best node in OPEN



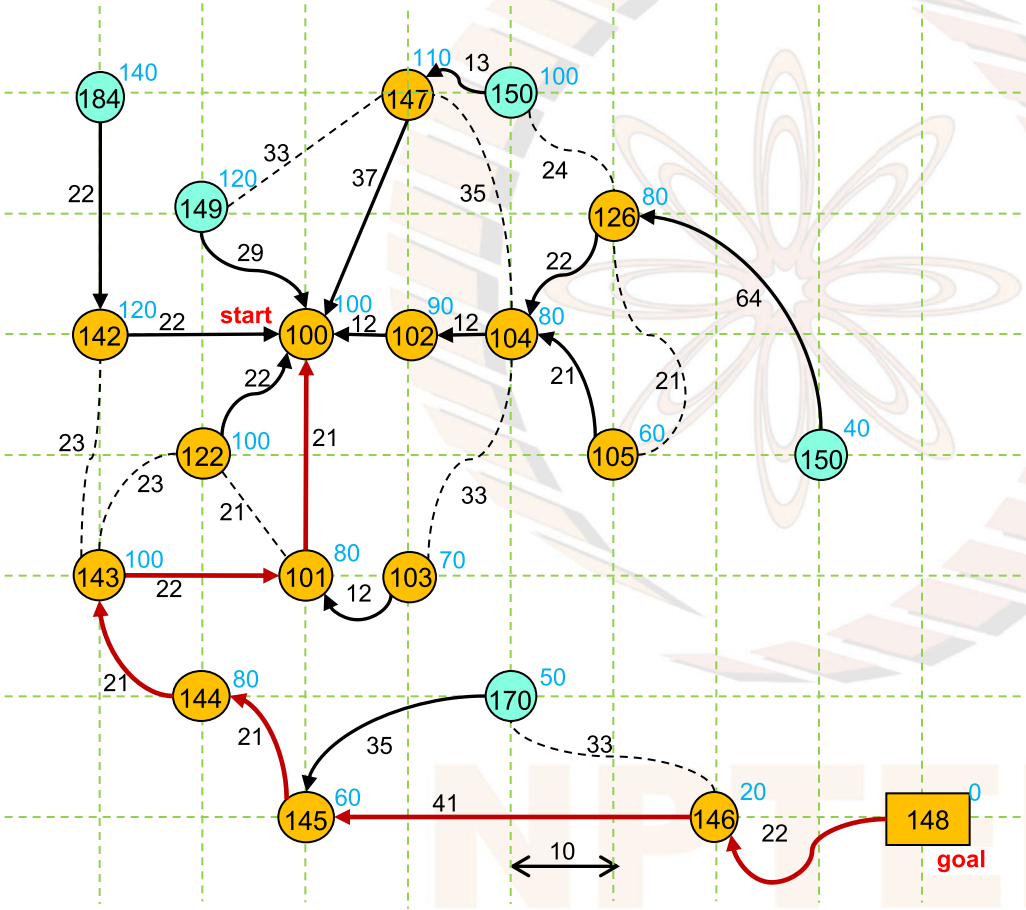
A*: Goal node generated but is there a better node?



A*: Goal node has lowest f-value

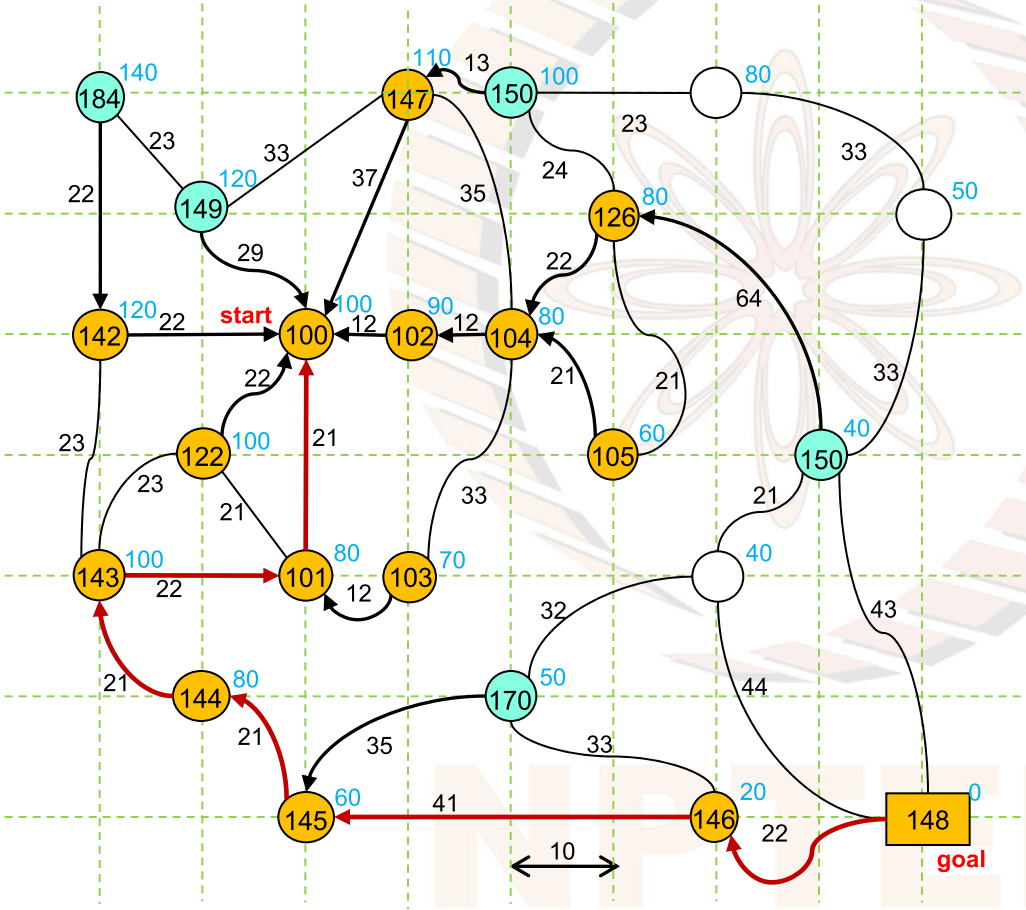


A*: The Solution



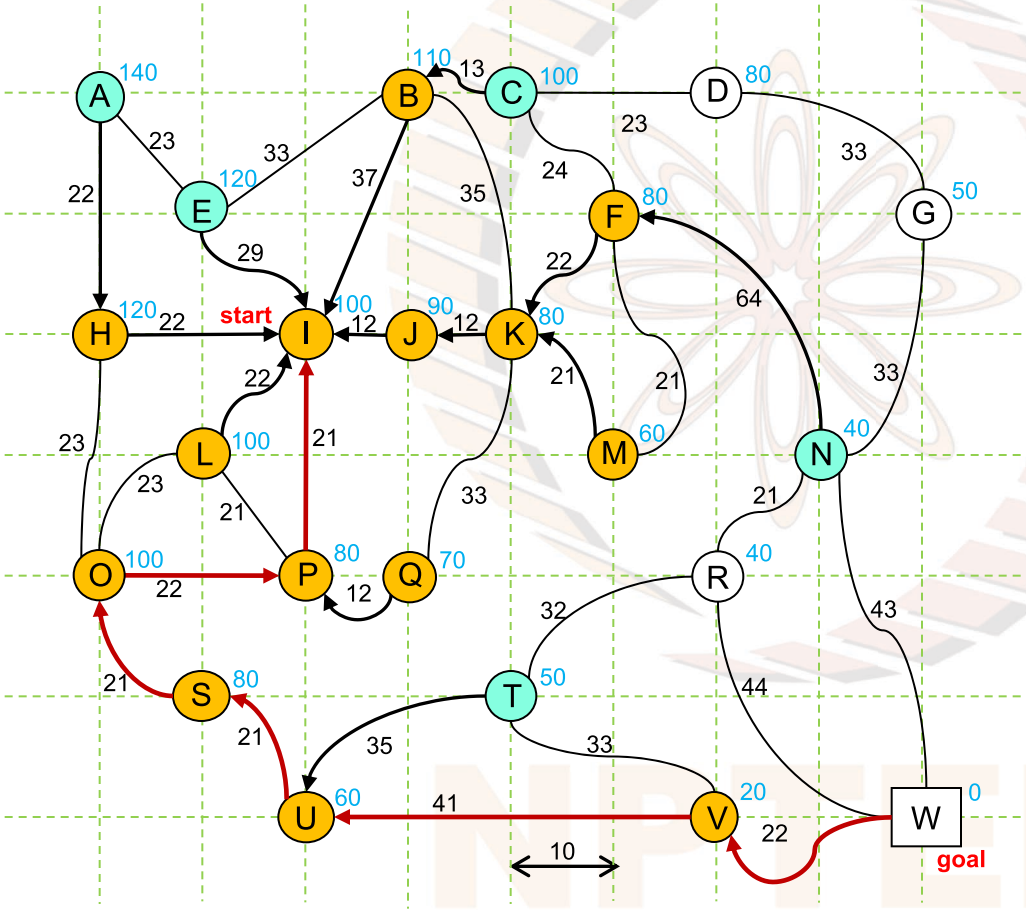
14 nodes inspected
Cost of solution = 148

A*: The nodes not explored



Exercise:
Try Dijkstra's algorithm

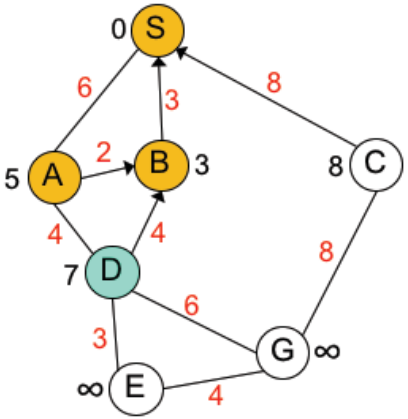
A*: The Solution : I,P,O,S,U,V,W



Next

Admissibility of A*

Dijkstra's Shortest Path Algorithm



Dijkstra's algorithm finds shortest paths to all nodes in the graph.

In the diagram on the right the algorithm has coloured the following nodes "black" (added them to CLOSED)

- S with cost 0
- B with cost 3
- A with cost 5 (via B)

At the point when the algorithm is about to pick the node D

- It is the cheapest node that has not been picked up
- Cost of D is 7 (via B) and it is the cheapest path to D
- No other cheaper path is possible for *any node* in OPEN
- Next cheapest path is to C with cost 8, and paths to other nodes can only be higher than 7.

A* is designed to find *an optimal path to the goal*

Given the algorithm as described *it appears* that A* *may not have found the cheapest path to a node* it picks up.

This is *suggested* by the fact that the Case 3 in the algorithm *caters to* finding *cheaper paths to nodes already in CLOSED*.

This is because A* works with *estimated costs* f(n) = g(n) + h(n) while Dijkstra's algorithm works with *known costs* g(n) to node *n*.

The estimate is of the *cost of going from Start to Goal* via node *n.*

But does A* *always* find the cheapest path to the goal node?
And does it always *find* a path if there exists one?
- *even* for an infinite graph?

Admissibility of A*

A search algorithm is said to be *admissible* if it is guaranteed to return an optimal solution if there exists one.

Some terminology

f*(n) = g*(n) + h*(n), where

- g*(n) is the optimal path cost from the Start node S to node *n*
- h*(n) is the optimal path cost from the node *n* to a goal node G
- f*(n) is the optimal cost of a path from S to G *via* node *n*

Note that these values are not known in general.

Observe that

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

because the algorithm may not have found the optimal path

Conditions for Admissibility of A*

What should the *heuristic function* be like if A* is to *always* find a least cost path?

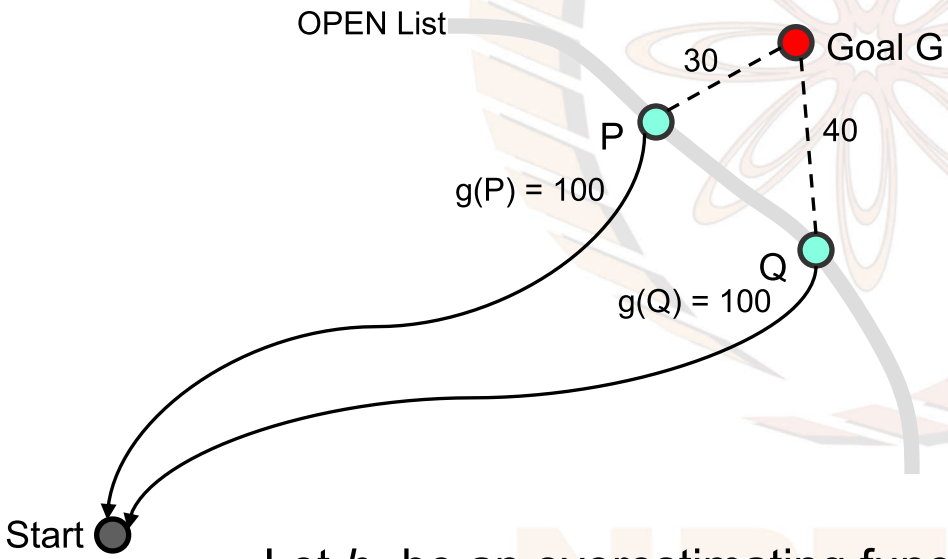
Assuming that a *perfect* heuristic function with h(n) = h*(n) does not exist, there are essentially two possibilities

1. For every node N the heuristic function consistently *underestimates* the distance to the goal, *i.e.* $h(n) \leq h^*(n)$
2. For every node N the heuristic function consistently *overestimates* the distance to the goal *i.e.* $h(n) \geq h^*n)$

Let us get some insight into this by a carefully constructed example.

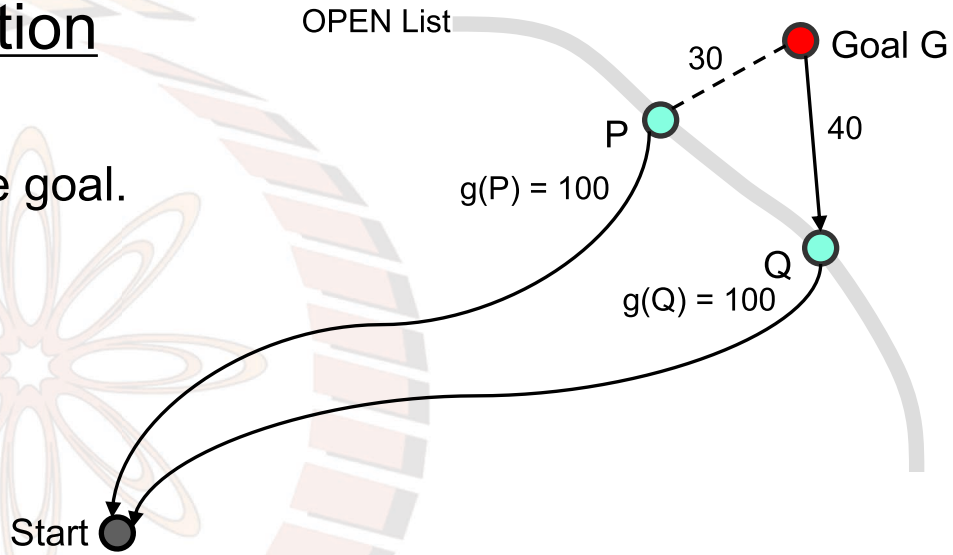
Overestimate or underestimate?

Consider the following example in which the algorithm is just one step away from the goal node, and has to choose between expanding P or expanding Q, both of which have g-value 100. Let cost(P,G) = 30 and cost(Q,G) = 40



Let h_1 be an overestimating function and let h_2 be an underestimating one.

An overestimating heuristic function



Let h_1 be an overestimating function and
let it misjudge the relative distance to the goal.

It thinks *Q* is closer than *P* to the goal.

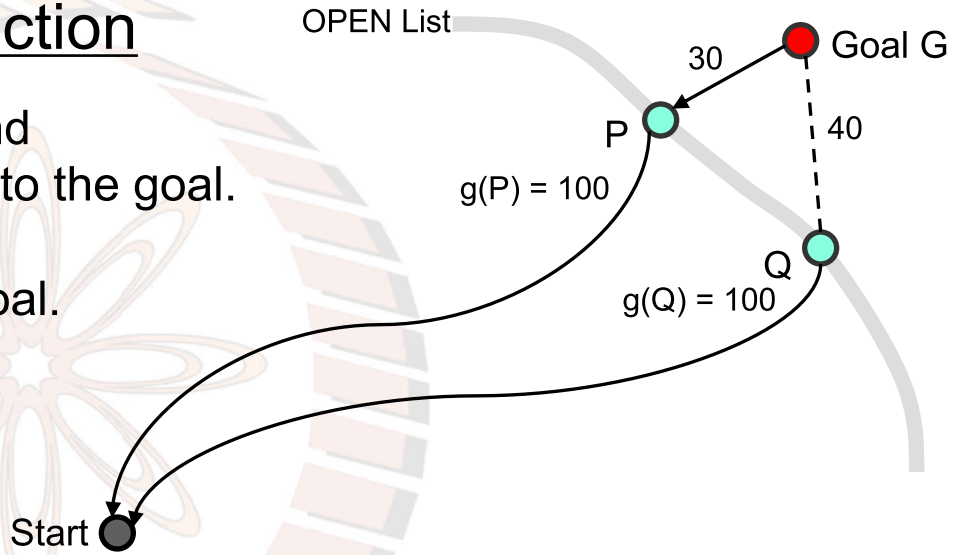
h(P) = 60, h(Q) = 50

therefore
f(P) = g(P) + h(P) = 100 + 60 = 160
f(Q) = g(Q) + h(Q) = 100 + 50 = 150

The algorithm picks *Q* and adds *G* to OPEN with g(G) = 140
f(G) = g(G) + h(G) = (100+40) + 0 = 140

Now it picks *G* with g(G) = 140 and terminates

An underestimating heuristic function



Let h_2 be an underestimating function and
let it ***also*** misjudge the relative distance to the goal.

It ***also*** thinks *Q* is closer than *P* to the goal.

h(P) = 20, h(Q) = 15

therefore
f(P) = g(P) + h(P) = 100 + 20 = 120
f(Q) = g(Q) + h(Q) = 100 + 15 = 115

The algorithm ***also*** picks *Q* and adds *G* to OPEN with g(G) = 140
f(G) = g(G) + h(G) = (100+40) + 0 = 140

*But **now** it picks P* and finds a shorter path to G with g(G)=130

A* is admissible if...

1. The branching factor is finite
   - otherwise you cannot even generate the neighbours
   - the *number* of nodes can still be infinite

2. Every edge has a cost greater than a small constant ε
   - Earlier literature only said that cost must be positive
   - But in the mid nineties a student in my class, Arvind Narayanan, pointed out that with positive costs it would be possible to have *an infinite path with finite cost* that the algorithm could get stuck in
   - For example with edge costs 1, ½, ¼ ... would add up to 2

3. For all nodes $h(n) \leq h^*(n)$
   - that is, the heuristic function underestimates the distance to the goal from *each* node *n*

We present the proof of admissibility through a series of lemmas

L1: The algorithm *always terminates* for *finite* graphs.

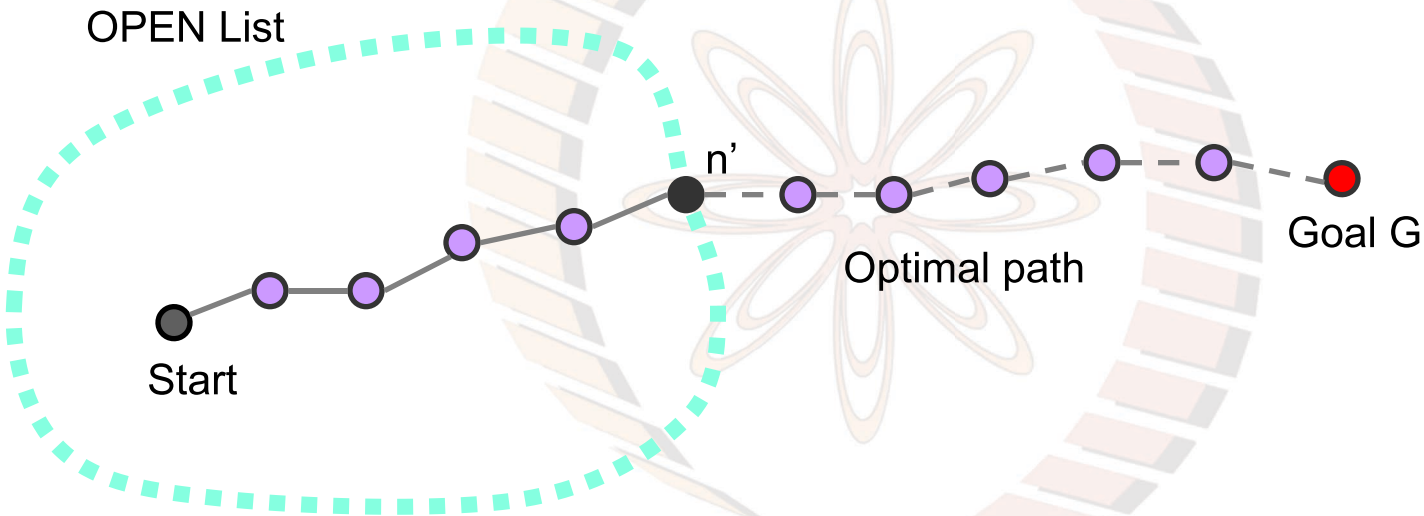
Proof: In every cycle of the main loop in *A** the algorithm picks one node from *OPEN* and places it in *CLOSED*.

Since there are only a finite number of nodes, the algorithm will terminate in a finite number of cycles.

If it finds a path to the goal it will report it,
and if it does not
it will (correctly) report that there is no path to the goal.

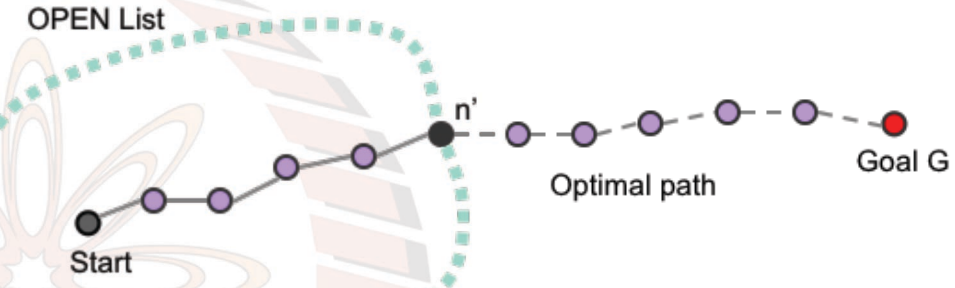
If the graph is finite the algorithm searches, if need be,
the *entire* connected component in which the start state is.

L2: OPEN always has node from the optimal path



If a path exists to the goal node, then the *OPEN* list always contains a node *n*' from an optimal path. Moreover the f-value of that node is not greater that the optimal cost.

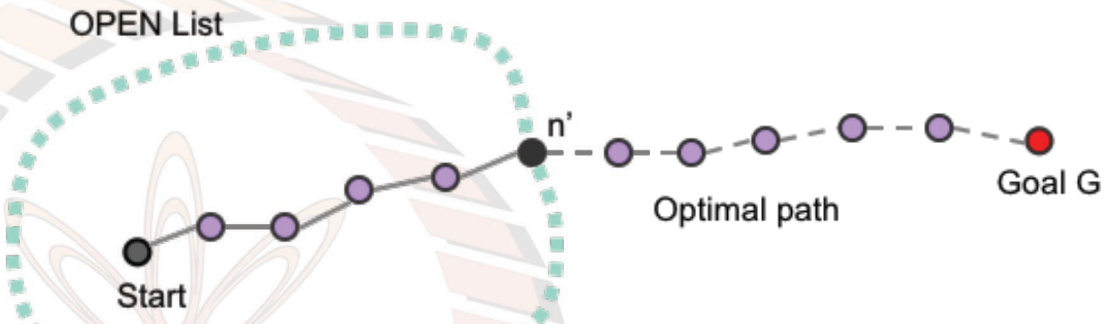
L2: OPEN always has node from the optimal path



Proof: Let P = (S, n_1, n_2, …, G) be the optimal path

- Initially S is on OPEN.
- When any node from P is removed from OPEN it's successor is added to OPEN.
- When G is removed the algorithm has terminated.

L2: ... and f(n') ≤ f*(S)



Furthermore,

f(n') = g(n') + h(n')
= g*(n') + h(n') because *n'* is on the optimal path g(n') = g*(n')
≤ g*(n') + h*(n') because h(n') ≤ h*(n')
≤ f*(n')
≤ f*(S) because *n'* is ***on*** the optimal path
f*(n') = f*(S)

∴ f(n') ≤ f*(S)

L3: *A** finds a path if there exists one...

... even if the graph is infinite

*Proof: A** always picks a node with the lowest f-value.

Every time it extends a partial solution, the g-value of the partial-solution increases by a finite value greater than ε.

Also since the branching factor is finite, there are only a finite number of partial solutions cheaper than the cost of a path to the goal, that is g(Goal).

After exploring all of them in a finite amount of time, eventually the path to the goal becomes cheapest, and is examined by *A** which then terminates with a path to the goal.

L4: *A** finds the least cost path to the goal

Proof: (by contradiction)
Assumption A4: Let *A** terminate with node *G'* with cost g(*G'*) > f*(*S*).

•When *A** was about to expand *G'* there existed a node *n'* such that f(n') ≤ f*(S) ----- by Lemma 2.

•Therefore f(n') < f(*G'*), and *A** would have picked *n'* instead of *G'*.

•Thus assumption A4 is wrong.

•Therefore, *A** terminates by finding the optimal cost path.

L5: For every node *n* expanded by *A**, f(n) ≤ f*(S)

... not just nodes on the optimal path

*Proof: A** picked node *n* in preference to node *n'*.

Therefore,

$$f(n) \leq f(n') \leq f^*(S)$$

More informed heuristic functions

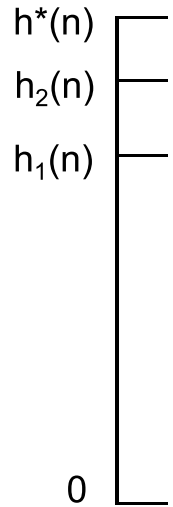
Let A_1 and A_2 be two admissible versions of A^*
using heuristic functions h_1 and h_2 respectively.

Let $h_2(n) > h_1(n)$ for all n.

Since both heuristic functions are admissible,
both heuristic functions have $h^*(n)$ as the upper bound.

We say h_2 is *more informed* than h_1,
because it is closer to the h^* value.

Both versions of A* will find an optimal path,
but A_2 will do so faster.



L6: More informed means less search

Let A_1 and A_2 use heuristic functions h_1 and h_2 respectively, such that

$\forall n(h_2(n) > h_1(n))$.

Then A_1 expands any node expanded by A_2. Formally,

$\forall n$ (Expands(A_2,n) $\supset$ Expands(A_1,n))

Proof: (by induction)

Basis: (Expands(A_2,S) $\supset$ Expands(A_1,S))

Induction hypothesis: Let it be true for all nodes till depth *K*

Induction step: Show for depth (K+1) ---- by contradiction

L6: ∀n (Expands(A_2,n) ⊃ Expands(A_1,n))

Assumption A6: Let there exist a node *L* at depth K+1 that is expanded by A_2, and such that A_1 terminates without expanding node *L*.

Since A_2 has picked node *L*,

$f_2(L) \leq f^*(S)$ by Lemma 5

That is $g_2(L) + h_2(L) \leq f^*(S)$, or

$h_2(L) \leq f^*(S) - g_2(L)$ (1)

Now since A_1 terminates without picking node *L*,

$f^*(S) \leq f_1(L)$ because otherwise A_1 would have picked *L*

or $f^*(S) \leq g_1(L) + h_1(L)$

or $f^*(S) \leq g_2(L) + h_1(L)$ because $g_1(L) \leq g_2(L)$

since A_1 has seen all nodes up to depth k seen by A_2,

and would have found an equal or better cost path to *L*.

or $f^*(S) - g_2(L) \leq h_1(L)$ (2)

L6: ∀n (Expands(A_2,n) ⊃ Expands(A_1,n))

Assumption A6: Let there exist a node *L* at depth K+1 that is expanded by A_2, and such that A_1 terminates without expanding node *L*.

From $h_2(L) \leq f^*(S) - g_2(L)$ (1)
and $f^*(S) - g_2(L) \leq h_1(L)$ (2)

we get $h_2(L) \leq h_1(L)$

which contradicts the given fact $\forall n(h_2(n) > h_1(n))$.

Ergo, assumption A6 is false, and A_1 would have expanded *L* too.

Therefore, ∀n(Expands(A_2,n) ⊃ Expands(A_1,n))

Next

Variations on A*

Leaner, meaner versions...