CSL020U4E: Artificial Intelligence
Lecture-05 (A^*-Algorithm)

Sumit Kumar Pandey

September, 2025

Branch and Bound

Branch and Bound (B&B)

Algorithm B&B searches in the space of partial paths. It extends the cheapest partial path till it finds one to the goal.

B&B-FirstCut(S)
  OPEN $\leftarrow$ ([S], 0) : [ ]
  while *OPEN* **is not empty**
    pathPair $\leftarrow$ **head** *OPEN*
    (path, cost) $\leftarrow$ *pathPair*
    $N \leftarrow$ **head** *path*
    if GoalTest(N) = true
      then return reverse(*path*)
    else
      children $\leftarrow$ MoveGen(N)
      newPaths $\leftarrow$ MakePaths(*children*, *pathPair*)
      OPEN $\leftarrow$ Sort$_{\textbf{cost}}$(*newPaths*++ **tail** *OPEN*)
  return empty list

Branch and Bound (B&B)

MakePaths(*children*, *pathPair*)
  if *children is empty*
    then return empty list
  else
    (*path*, *cost*) $\leftarrow$ *pathPair*
    $M \leftarrow$ **head** *children*
    $N \leftarrow$ **head** *path*
    return ([M:*path*], cost + $k(N, M)$): MakePaths(**tail** *children*, *pathPair*)

Branch and Bound (B&B)

A version of B&B that avoids getting into loops. If a child of N is already present in the path, then that is discarded.

```
B&B(S)
  OPEN ← ([S], 0) : [ ]
  while OPEN is not empty
    pathPair ← head OPEN
    (path, cost) ← pathPair
    N ← head path
    if GoalTest(N) = true then return reverse(path)
    else
      children ← MoveGen(N)
      noloops ← RemoveSeen(children, path)
      newPaths ← MakePaths(noloops, pathPair)
      OPEN ← Sort_cost(newPaths++ tail OPEN)
  return empty list
```

Branch and Bound (B&B)

```
RemoveSeen(children, path)
  if children is empty then return empty list
  else
    M ← head children
    if OccursIn(M, path)
      then return RemoveSeen(tail children, path)
    else return M:RemoveSeen(tail children, path)

OccursIn(node, list)
  if list is empty then return False
  else if node == head list then return True
    else return OccursIn(node, tail list)
```

Performance of B&B

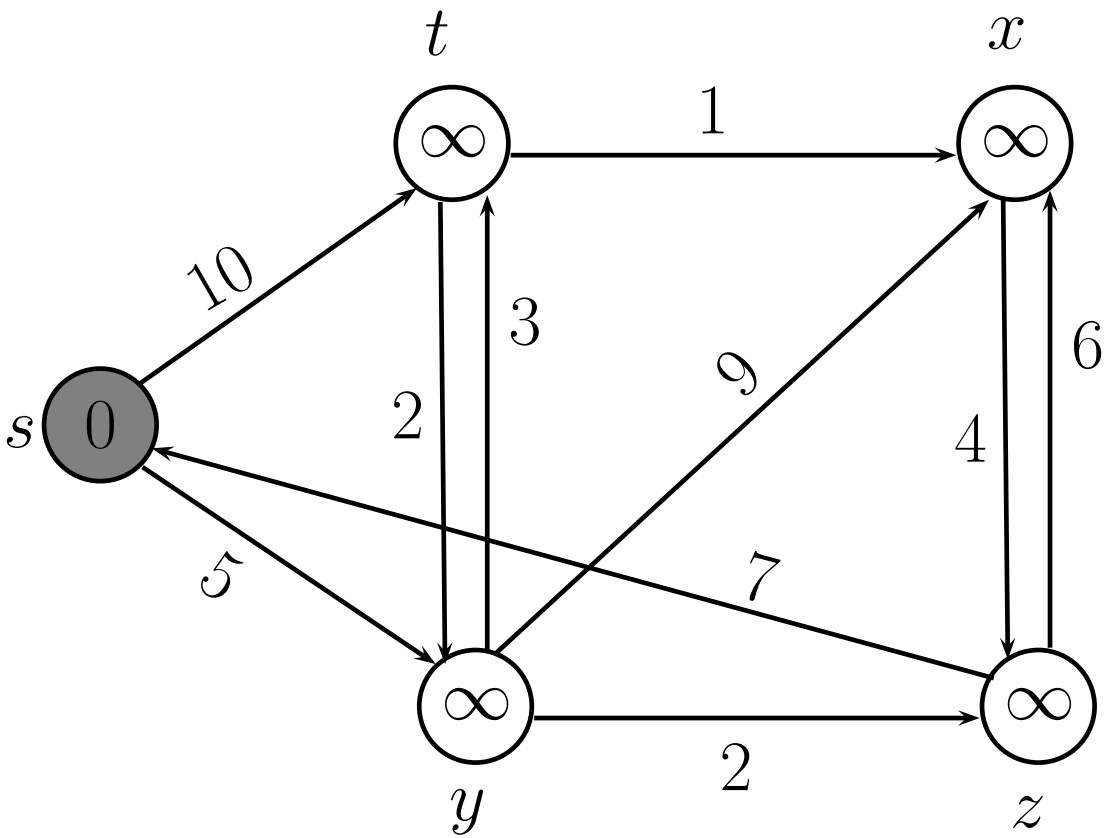
1. Completeness: The algorithm is complete. It will even terminate on infinite graphs if there is a path from the start node to the goal and each edge has a cost greater than some $\epsilon > 0$. This is because it explores paths in increasing order of cost and will eventually find a path to a goal node.
2. Quality of solution: For the same reason, it always finds the optimal path to the goal node. This is because it picks the least cost path to goal first.

Performance of B&B

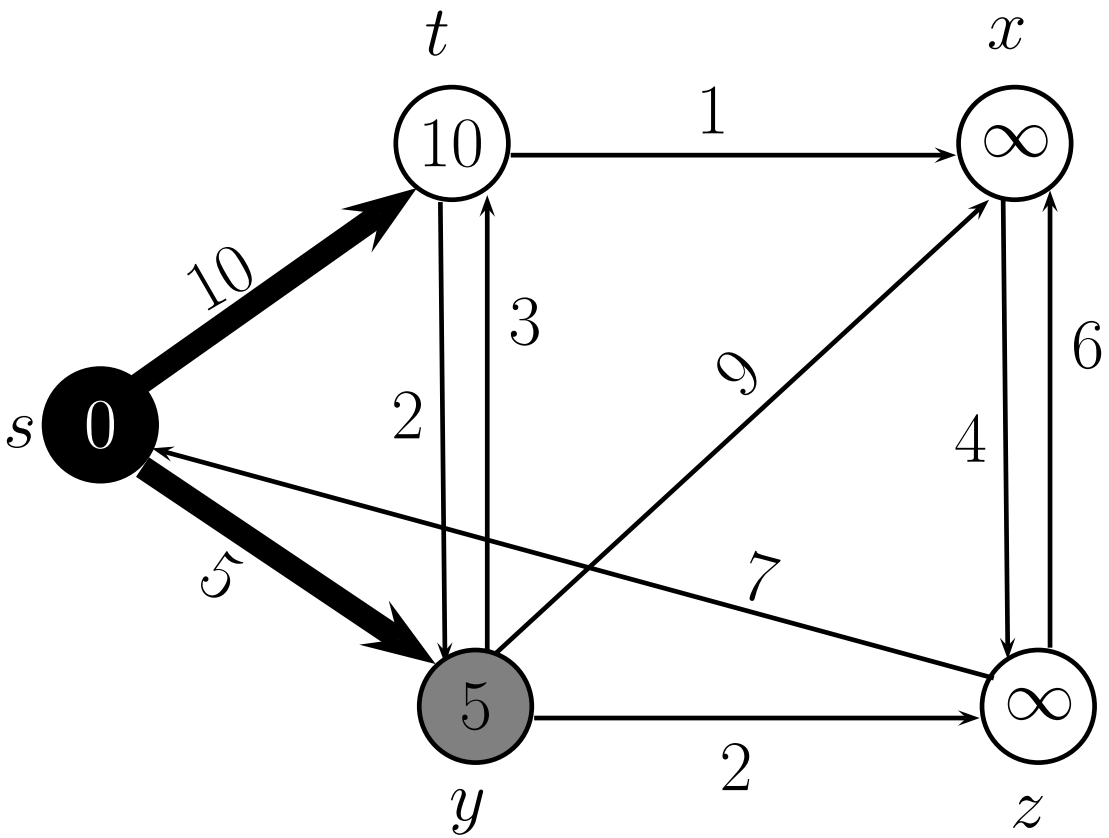
3. Space complexity: The algorithm needs exponential space since it keeps all partial paths in a tree at all times. In each cycle, it extends the cheapest partial path.
4. Time complexity: Time complexity is exponential in depth. This is because it combinatorially explores all partial paths of cost less than the optimal path to the goal.

Dijkstra's Algorithm

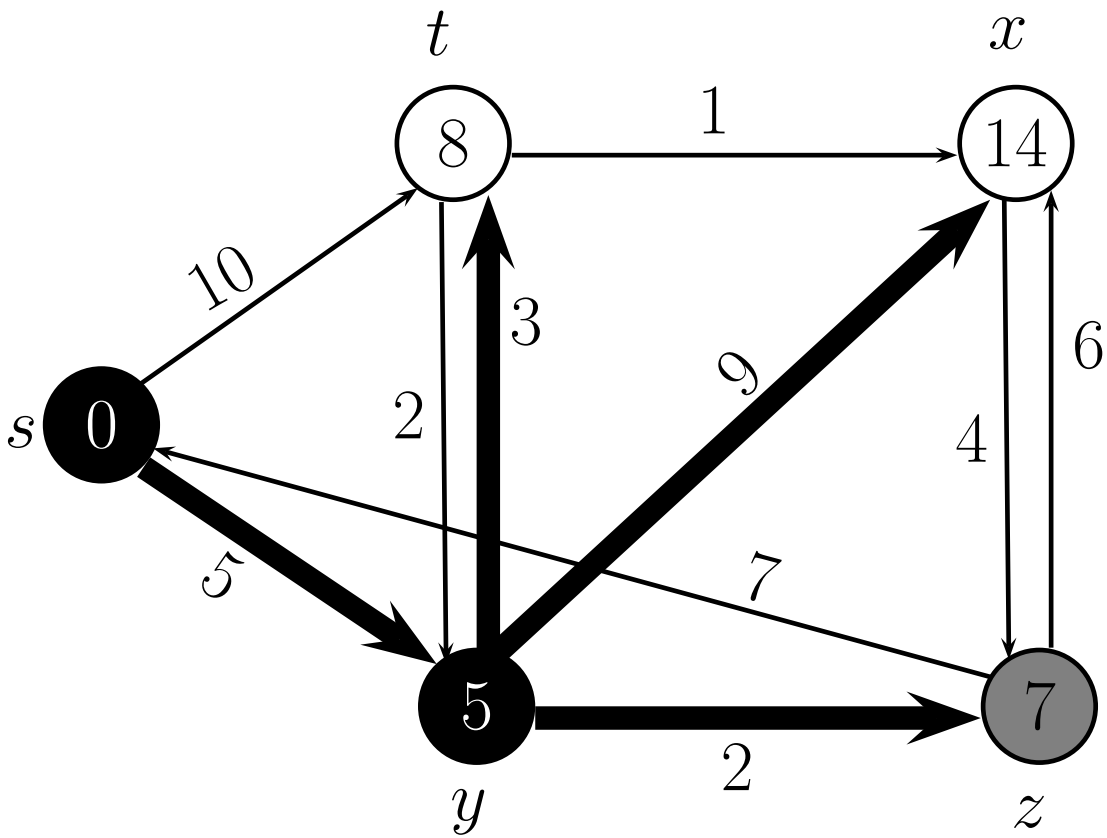
Dijkstra's Algorithm



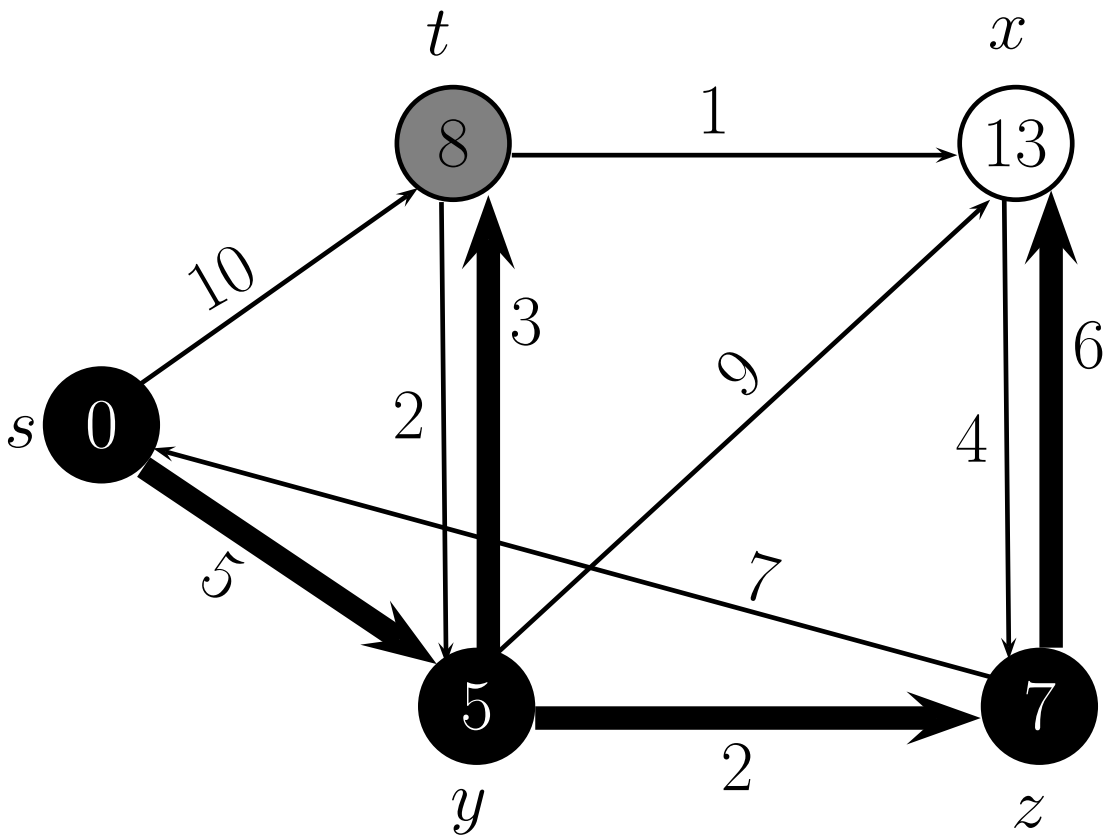
Dijkstra's Algorithm



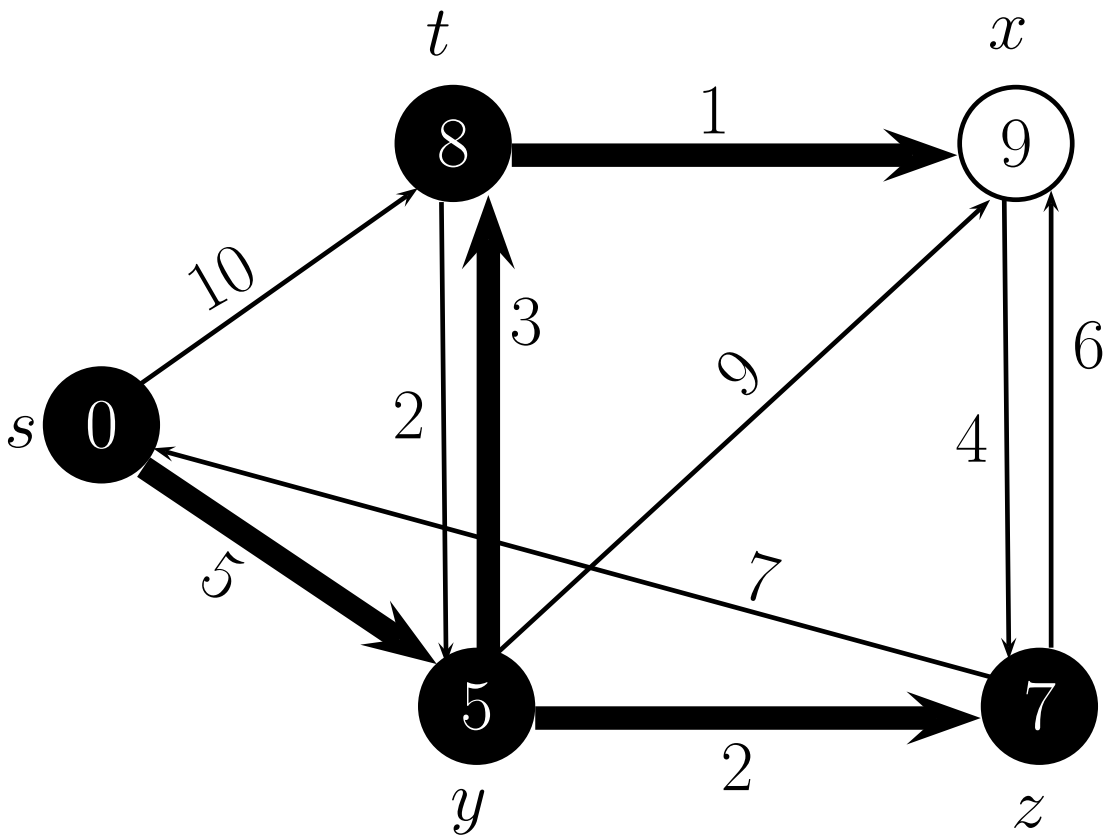
Dijkstra's Algorithm



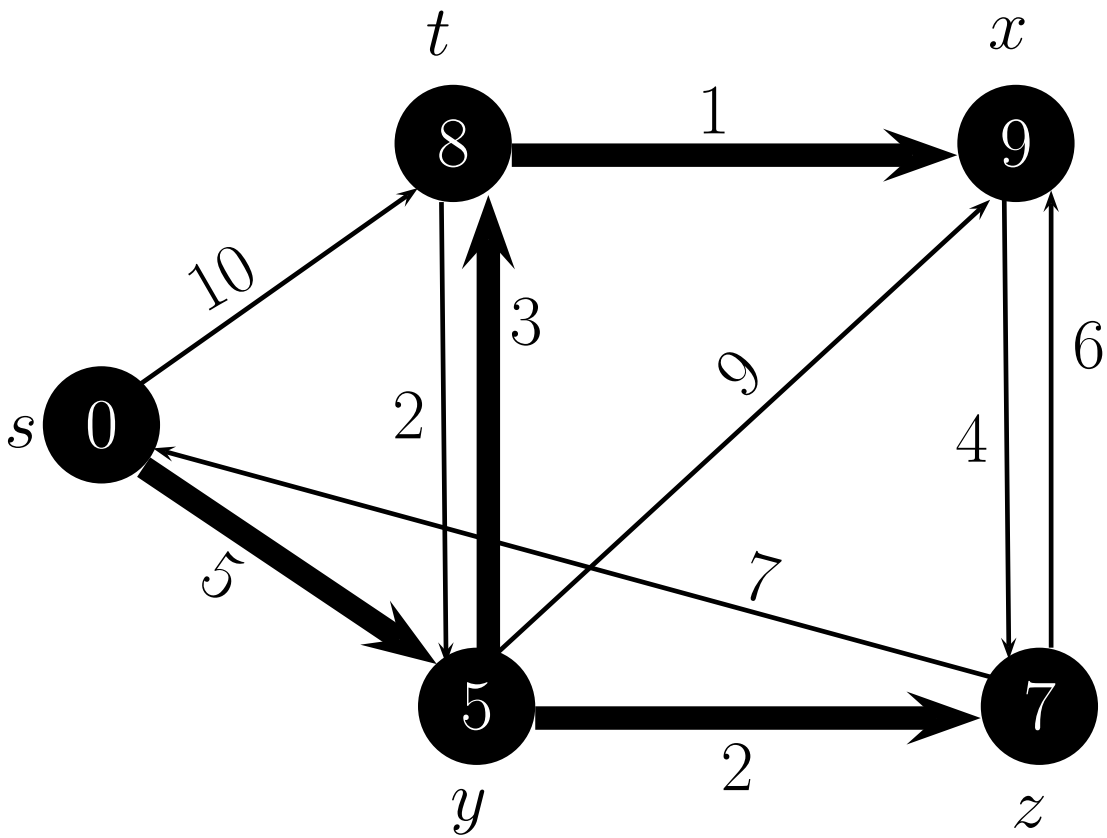
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



Algorithm A^*

*Algorithm A**

- The algorithm A^*, first described by Hart, Nilsson, and Raphael (Hart, Nilsson, and Raphael, 1968; Nilsson, 1971, 1980) is a heuristic search algorithm that has found wide-ranging applications.

Algorithm A^* combines several features of the algorithms we have studied so far:

1. BestFirstSearch
2. B&B
3. Dijkstra

*Algorithm A**

Algorithm A^* is

1. Like BestFirstSearch it employs a heuristic function $h(n)$ that is an estimate of the distance to the nearest goal. This gives search a sense of direction, so that it is focussed on the goal. For simplicity, we will assume there is one goal node in our description.
2. Like B&B it keeps track of the known distance from the start node. This is done by a function named $g(n)$. Preferring low g-values results in a tendency to stay close to the start node. This is necessary since we are interested in shortest paths.
3. Like Dijkstra algorithm it works with a graph representation, keeping exactly one copy of every node. Associated with each node is a function $parent(n)$ that points to the parent node. Like Dijkstra's algorithm it marks the parent which is on the shortest path to n. The parent pointer is instrumental in reconstructing the path.

Algorithm A^*

Every node n in A^* has an estimated cost $f(n)$ of the path from the start node to the goal node passing through n.

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

That is, $f(n)$ is the estimated cost of the solution containing node n. The algorithm always refines or expands the node with the lowest f-value.

Like our earlier algorithms A^* picks the best node n from *OPEN* and checks whether it is the goal node. If it is, then it follows the pointers back to reconstruct the path. If it is not, it adds the node to *CLOSED* and generates its neighbours.

*Algorithm A**

For each neighbour m of n it does the following:

1. If m is a new node, it adds it to *OPEN* with parent n and $g(m) = g(n) + k(n, m)$ where $k(n, m)$ is the cost of the edge connecting m and n.
2. If m is on *OPEN* with parent n', then it checks if a better path to m has been found. If yes, then it updates $g(m)$ and sets its parent to n.
3. If m is on *CLOSED* with parent n', then it checks if a better path to m has been found. If yes, then it updates $g(m)$ and sets its parent to n. This possibility exists since $h(n)$ is an estimate and may be imperfect, choosing n' before n. The algorithm will also need to propagate this improved cost to other descendants of m.

A^* *is admissible*

A^* always finds an optimal path if there is a path from the start node to the goal node. This is true even if the graph is infinite under the following conditions:

1. The MoveGen or neighbourhood function has a finite branching factor.
2. The cost of every edge must be greater than a small constant ϵ. This, as we is to preclude the possibility of getting trapped in an infinite path with a finite total cost.
3. The heuristic function must underestimate the distance to the goal $h(n)$ for every node.

A^* Algorithm

$A^*(S)$
  parent(S) $\leftarrow$ **null**
  $g(S) \leftarrow 0$
  $f(S) \leftarrow g(S) + h(S)$
  $OPEN \leftarrow S : [\,]$
  $CLOSED \leftarrow$ **empty list**
  while $OPEN$ **is not empty**
    $N \leftarrow$ **remove node with lowest** f**-value from** $OPEN$
    add N to CLOSED
    if GoalTest(N) = TRUE **then return** ReconstructPath(N)
    for each neighbour $M \in$ MoveGen(N)
      if ($M \notin OPEN$ **and** $M \notin CLOSED$)
        parent(M) $\leftarrow N$
        $g(M) \leftarrow g(N) + k(N, M)$; $f(M) \leftarrow g(M) + h(M)$
        add M **to** $OPEN$
      else
        if $(g(N) + k(N, M)) < g(M)$
          parent(M) $\leftarrow N$
          $g(M) \leftarrow g(N) + k(N, M)$; $f(M) \leftarrow g(M) + h(M)$
          if $M \in CLOSED$
            PropagateImprovement(M)
  return empty list

A^* *Algorithm*

PropagateImprovement(M)
  for each neighbour $X \in$ MoveGen(M)
    if $g(M) + k(M, X) < g(X)$
      parent(X) $\leftarrow M$
      $g(X) \leftarrow g(M) + k(M, X)$
      $f(X) \leftarrow g(X) + h(X)$
      if $X \in$ *CLOSED*
        PropagateImprovement(X)

Proof for the admissibility of A^*

- Let $g^*(n)$ be the optimal path cost from the start node S to node n. Observe that this is not known in general. What we do know is $g(n)$ which is the cost of the path found by the algorithm.
- Observe that $g^*(n) \leq g(n)$ because the algorithm may not have found the optimal path.
- Let $h^*(n)$ be the optimal path cost from the node n to a goal node G. Again, this is not a known quantity.
- Then $f^*(n) = g^*(n) + h^*(n)$ is the optimal cost of a path from S to G *via* node n. This value is the same for any node on an optimal path.
- In particular, $f^*(S) = h^*(S)$ stands for the cost of the optimal path.

Proof for the admissibility of A^*

In addition, we have the conditions stated above for admissibility, repeated here.

1. For all n: $|\textsc{MoveGen}(n)| \leq b$ for some value of b - $C1$.
2. For every edge $\langle n, m \rangle$: $k(n, m) > \epsilon > 0$ - $C2$.
3. For all n: $0 \leq h(n) \leq h^*(n)$ - $C3$.

Given the above, we prove a series of lemmas leading to the proof of admissibility of A^*.

Proof for the admissibility of A^*

Lemma-L0

Let $(S, n_1, n_2, \ldots, G)$ be an optimal path, then

$$f^*(S) = f^*(n_1) = f^*(n_2) = \cdots = f^*(G)$$

Proof:

- Choose any node n belongs to the path $(S, n_1, n_2, \ldots, G)$. Consider two paths $P_1 = (S, n_1, \ldots, n)$ and $P_2 = (n, \ldots, G)$ where weight of path $P_1 = w_1$ and $P_2 = w_2$.
- Note that $g^*(n) = w_1$ and $h^*(n) = w_2$.
- So, $f^*(n) = g^*(n) + h^*(n) = w_1 + w_2 = w$, the weight of the path $(S, n_1, n_2, \ldots, G)$.
- Since it is true for any node in the path $(S, n_1, n_2, \ldots, G)$, hence the result.

*Proof for the admissibility of A**

Lemma - L1

The algorithm always terminates for finite graphs.

Proof:

- The algorithm keeps exactly one copy of every node generated, either in *OPEN* or in *CLOSED*.
- In every cycle, it picks one node from *OPEN* and moves it to *CLOSED* if it is not a goal node.
- Since the total number of nodes is finite, there will be eventually none left to add to *OPEN*. If the goal node is not in the graph, *OPEN* will become empty and the algorithm will terminate.

Proof for the admissibility of A^*

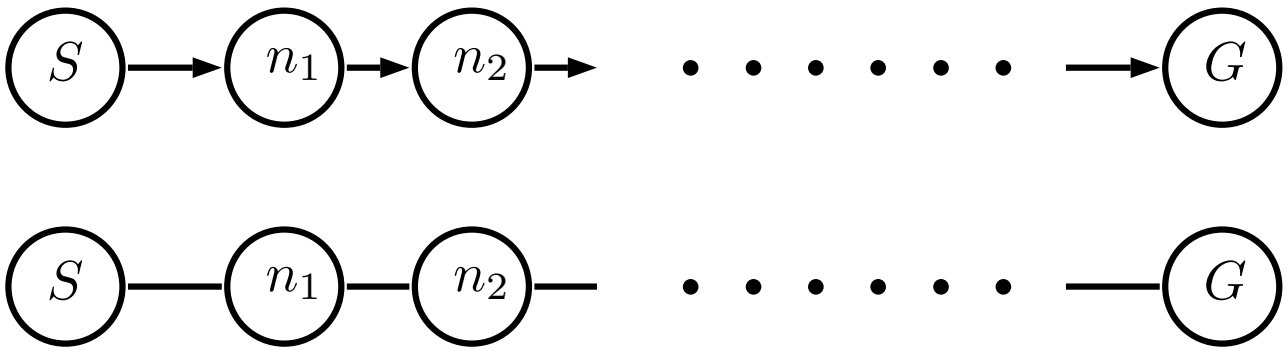
Lemma - L2

If a path exists from the start node to the goal node, then at all times before termination *OPEN* always has a node n' on an optimal path. Furthermore, $g(n') = g^*(n')$ and the f-value of this node is optimal or less, i.e. $f(n') \leq f^*(n')$.

Proof:

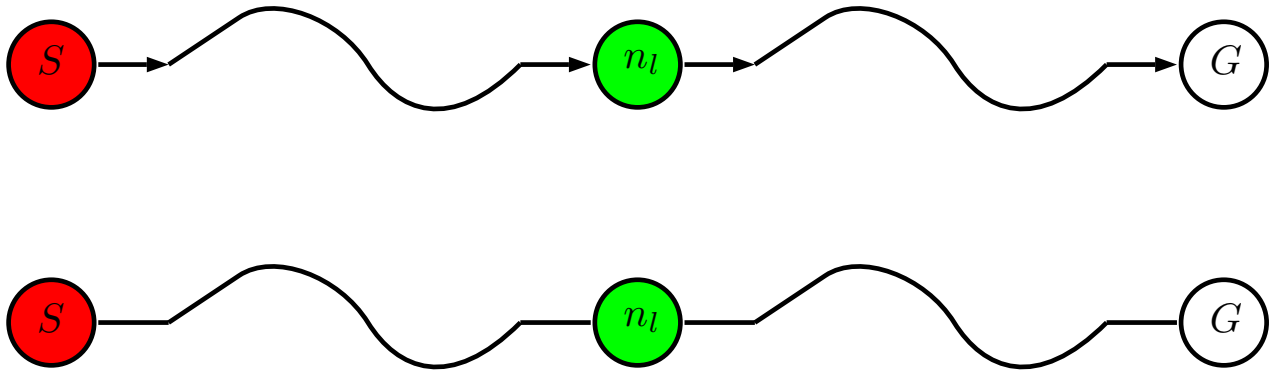
- Let $(S = n_0, n_1, n_2, \ldots, G)$ be such an optimal path. We now prove by induction on the index of nodes in the optimal path.
- Base step: The algorithm begins by adding S to *OPEN*. $S = n_0$ be in *OPEN* initially, $g(S) = g^*(S)$ and $f(S) = f(n_0) = g(n_0) + h(n_0) = 0 + h(n_0) \leq 0 + h^*(n_0) = g^*(n_0) + h^*(n_0) = f^*(n_0) = f^*(S)$.
- Induction hypothesis: At certain iteration of A^* algorithm, let the node n_l for $l \geq 0$ be in OPEN and (a) $g(n_l) = g^*(n_l)$ and (b) $f(n_l) \leq f^*(n_l)$.

*Proof for the admissibility of A**



OPTIMAL PATH

*Proof for the admissibility of A**



The Induction Hypothesis: The node n_l is in OPEN and (a) $g(n_l) = g^*(n_l)$ and (b) $f(n_l) \leq f^*(n_l)$.

*Proof for the admissibility of A**

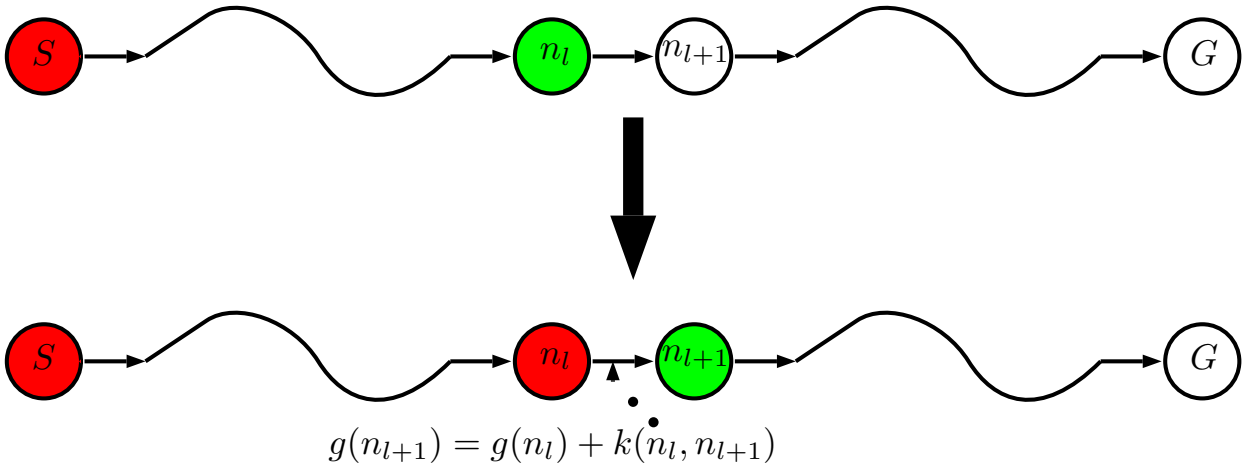
Proof (continued):

- Induction step: Let n_l be removed from *OPEN* and put it in *CLOSED*. The MoveGen function returns neighbours of the node n_l. One of the node will be n_{l+1} which is the successor node of node n_l in optimal path $(S, n_1, \ldots, G)$. There are two cases now:
  1. n_{l+1} is neither in *OPEN* nor in *CLOSED*. Then, n_{l+1} goes into *OPEN*.
     $g(n_{l+1}) = g(n_l) + k(n_l, n_{l+1}) = g^*(n_l) + k(n_l, n_{l+1})$. Since n_{l+1} is in the optimal path, so $g^*(n_l) + k(n_l, n_{l+1}) = g^*(n_{l+1})$. Thus, $g(n_{l+1}) = g^*(n_{l+1})$. Moreover,

$$\begin{aligned} f(n_{l+1}) &= g(n_{l+1}) + h(n_{l+1}) \\ &= g^*(n_{l+1}) + h(n_{l+1}) \\ &\leq g^*(n_{l+1}) + h^*(n_{l+1}) \qquad (C3). \\ &= f^*(n_{l+1}). \end{aligned}$$

     Take $n' = n_{l+1}$.

*Proof for the admissibility of A**



*Proof for the admissibility of A**

Proof (continued):

- Induction step (continued):
  2. n_{l+1} is in *OPEN*. Then, $g(n_{l+1})$ needs to be revised. New

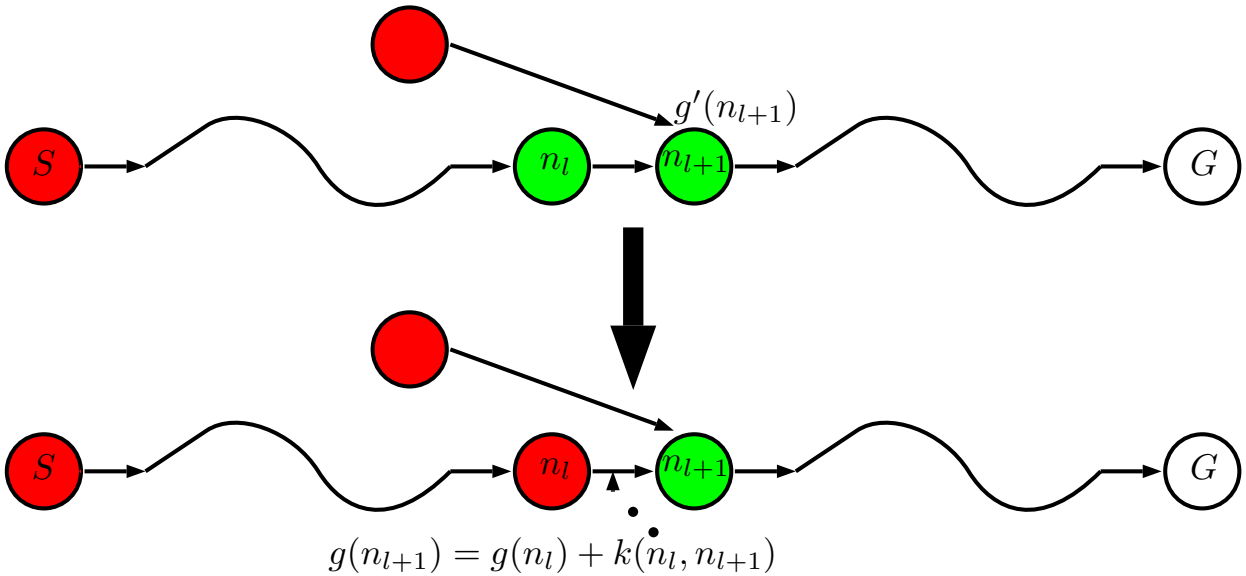
$$g(n_{l+1}) = g(n_l) + k(n_l, n_{l+1}) = g^*(n_{l+1})$$

and new

$$\begin{aligned} f(n_{l+1}) &= g(n_{l+1}) + h(n_{l+1}) \\ &= g^*(n_{l+1}) + h(n_{l+1}) \\ &\leq g^*(n_{l+1}) + h^*(n_{l+1}) \qquad (C3). \\ &= f^*(n_{l+1}). \end{aligned}$$

Take $n' = n_{l+1}$.

*Proof for the admissibility of A**



*Proof for the admissibility of A**

Proof (continued):

- Induction step (continued):
  3. n_{l+1} is in *CLOSED*. Then A^* algorithm revises its "g" value if required. One can observe that since n_{l+1} lies in the optimal path, so earlier value of $g(n_{l+1})$ must be higher than the new value. The new value is

$$g(n_{l+1}) = g(n_l) + k(n_l, n_{l+1}) = g^*(n_l) + k(n_l, n_{l+1}) = g^*(n_{l+1})$$

Then, $g(n_{l+1})$ needs to be revised. New

$$g(n_{l+1}) = g(n_l) + k(n_l, n_{l+1}) = g^*(n_{l+1})$$

and new

$$\begin{aligned} f(n_{l+1}) &= g(n_{l+1}) + h(n_{l+1}) \\ &= g^*(n_{l+1}) + h(n_{l+1}) \\ &\leq g^*(n_{l+1}) + h^*(n_{l+1}) \qquad (C3). \\ &= f^*(n_{l+1}). \end{aligned}$$

*Proof for the admissibility of A**

Proof (continued):

- Induction step (continued):
  - (continued) If n_{l+1} is in *CLOSED*, then, n_{l+1} was once in *OPEN* and when it was added in *CLOSED*, the node n_{l+2} was added in *OPEN*. and so on. We find
    $r = \min_{i>l+1}\{n_i \text{ is in } OPEN \text{ and } n_i \text{ is in } (S, n_1, n_2, \ldots, G)\}$.
    From previous arguments,

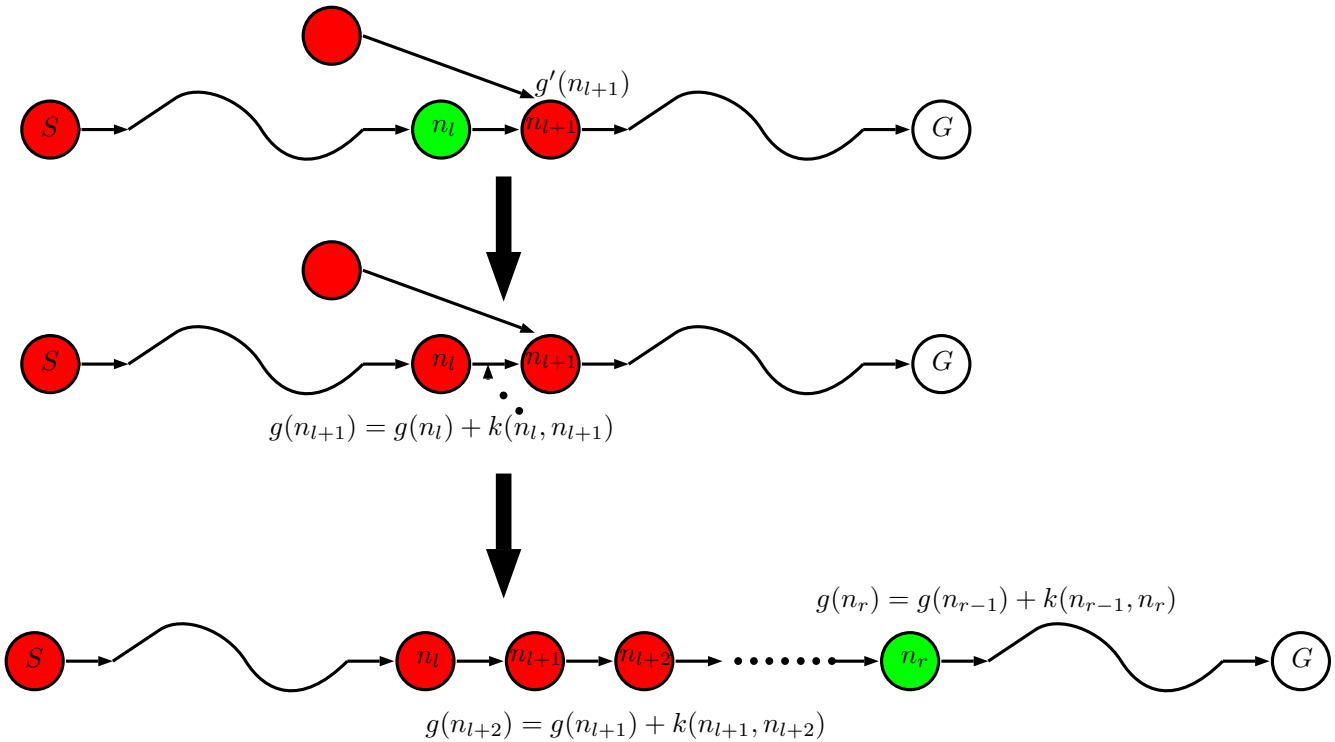
    $$g(n_r) = g(n_{r-1}) + k(n_{r-1}, n_r) = g^*(n_{r-1}) + k(n_{r-1}, n_r) = g^*(n_r)$$

    and,

    $$\begin{aligned} f(n_r) &= g(n_r) + h(n_r) \\ &= g^*(n_r) + h(n_r) \\ &\leq g^*(n_r) + h^*(n_r) \qquad (C3). \\ &= f^*(n_r). \end{aligned}$$

    Take $n' = n_r$.

*Proof for the admissibility of A**



Proof for the admissibility of A^*

Lemma - L3

A^* terminates even if the graph is infinite.

Proof: We prove it by showing that

- If there is any node which is in the optimal path and in *OPEN* will be eventually picked and added into *CLOSED*.
- And if it is true, either all nodes from the optimal path will be picked **or** at some itertaion a goal node is picked without picking all nodes from the optimal path.
- In either case, a goal node is picked and the algorithm will terminate.

So, we prove now

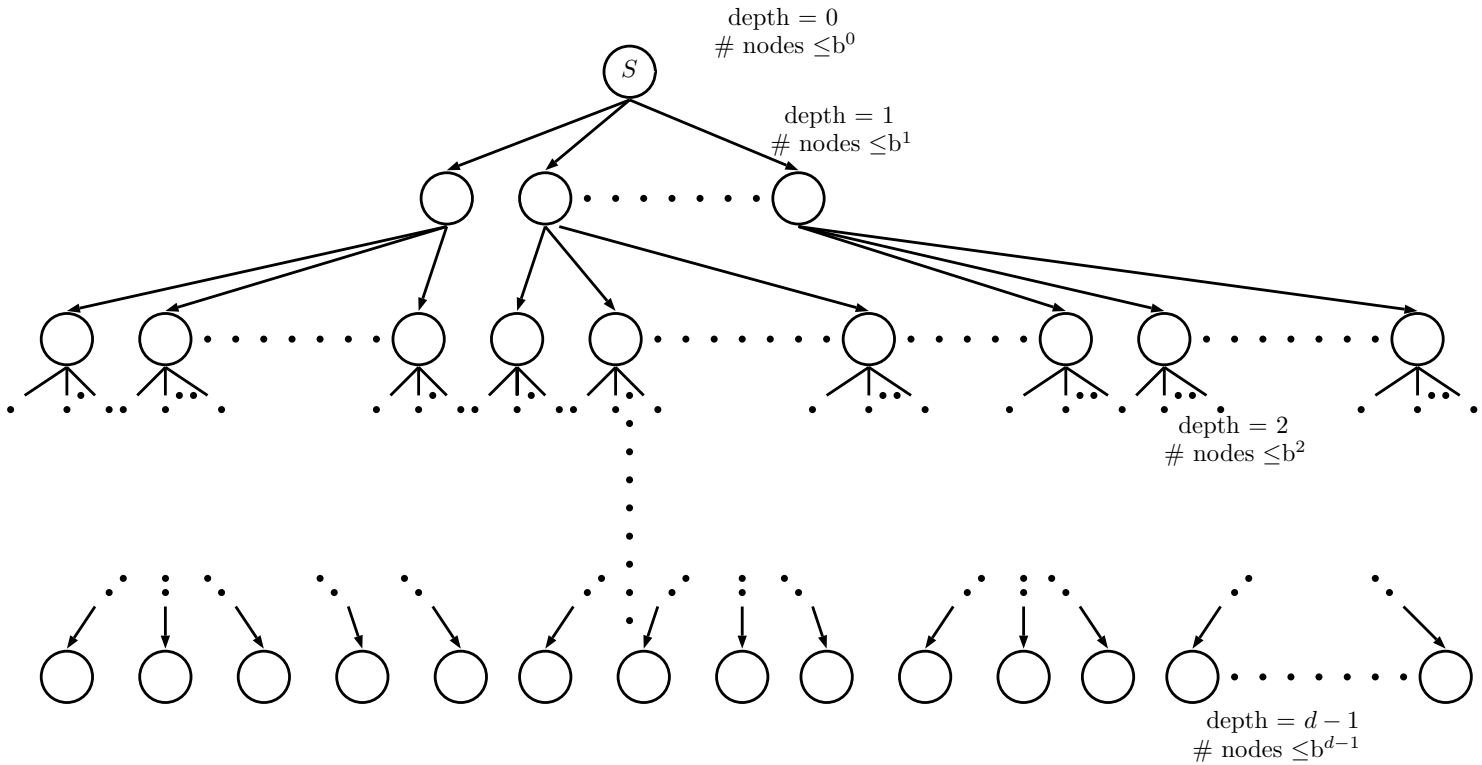
If there is any node which is in the optimal path and in *OPEN* will be eventually picked and added into *CLOSED*.

Proof for the admissibility of A^*

Proof (continued):

- If $b = 1$, then there exists a unique path from the startnode to the goalnode and can be found in finite time. So, we assume $b \geq 2$.
- Since the branching factor is less than or equal to b (a finite number), there exists a path of length d if b^d nodes are chosen. The reason is
  - At depth 0, there exists $b^0 = 1$ node. At depth 1, there exists at most b nodes. Similarly, at depth $d-1$, there exists at most b^{d-1} nodes.
  - So there can be at most $b^0 + b^1 + \cdots + b^{d-1} = (b^d - 1)/(b-1) < b^d$ nodes.
  - So, if b^d nodes are chosen, there will certainly be a path of length d. Let's call this path $\mathcal{P}$.
- Since each edge has weight greater than $\epsilon > 0$, so there exists a path of weight d.

*Proof for the admissibility of A**



Proof for the admissibility of A^*

Proof (continued):

- From Lemma-L2, there exists a node n' such that n' is in optimal path and n' is in *OPEN*.
- Choose $d > f(n')/\epsilon$ so that $\epsilon d > f(n')$.
- The last node in the path $\mathcal{P}$ will have f value greater than or equal to ϵd which is greater than $f(n')$, a contradiction, beacuse A^* chooses a node form *OPEN* with the least f value.
- So, n' from *OPEN* will be eventually picked and added into the *CLOSED*.

Proof for the admissibility of A^*

Lemma - L4

A^* finds the least cost path to the goal.

Proof (by contradiction): Let A^* terminate with node G', instead of optimal G, with cost $g(G') = f(G') > f(G) = f^*(G) = f^*(n')$.

- This could not happen. Because by L2 and L0, there is always a cheaper node n', i.e. $f(n') \leq f^*(n') = f^*(G) < f(G')$ in *OPEN* that is on the optimal path.
- It would instead pick n'. Therefore, A^* terminates by finding the optimal cost path.

Lemma- L5

For every node n expanded by A^*, $f(n) \leq f^*(S)$.

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

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

Proof for the admissibility of A^*

Lemma - L6

Let A_1^* and A_2^* be two admissible versions of A^* and let $h^*(n) > h_2(n) > h_1(n)$ for all n. We say h_2 *is more informed than* h_1, because it is closer to the h^* value. Then the search space explored by A_2^* is contained in the space explored by A_1^*. Every node visited by A_2^* is also visited by A_1^*. Whenever a node is visited it is also added to *CLOSED*.

Proof (by induction): To show that if $n \in CLOSED_2$, then $n \in CLOSED_1$.

- Base step: If $S \in CLOSED_2$, then $S \in CLOSED_1$.
- Induction hypothesis: Let the statement be true for all nodes upto depth d. If $n \in CLOSED_2$, then $n \in CLOSED_1$.

Proof for the admissibility of A^*

- Induction step (by contradiction): Let L be a node at depth $(d+1)$ such that $L \in CLOSED_2$ and let A_1^* terminate without inspecting L. Since A_2^* has picked node L,

$$\begin{array}{rrcll} & f_2(L) & \leq & f^*(S) & \text{from } L5 \\ \text{That is} & g_2(L) + h_2(L) & \leq & f^*(S) & \\ \text{or} & h_2(L) & \leq & f^*(S) - g_2(L) & \end{array}$$

- Since A_1^* terminates without picking node L,

$$\begin{array}{rcl} f^*(S) & \leq & f_1(L) \text{ because otherwise } A_1^* \text{ would have picked } L \\ \text{or } f^*(S) & \leq & g_1(L) + h_1(L) \\ \text{or } f^*(S) & \leq & g_2(L) + h_1(L) \end{array}$$

  because $g_1(L) \leq g_2(L)$ since A_1^* has seen all nodes up to depth d seen by A_2^*, and would have found an equal or better cost path to L.

Proof for the admissibility of A^*

- We can rewrite the last inequality as

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

  We already have $h_2(L) \leq f^*(S) - g_2(L)$.
- Combining the above two, we get

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

  which contradicts the given fact that $h_2(n) > h_1(n)$ for all nodes.
- The assumption that A_2^* terminates without expanding L must be false, and therefore A_2^* must expand L. Since L was an arbitrary node picked at depth $(d+1)$, the following is true at depth $d+1$ as well:

$$\text{If } n \in CLOSED_2, \text{ then } n \in CLOSED_1.$$

  Hence by induction, it is true at all levels.

The Monotone Condition

- When Dijkstra's algorithm picks a node from *OPEN*, it has already found the optimal path to that node.
- In contrast, when A^* picks a node and adds it to *CLOSED*, it may yet find another shorter path to it. That is why the algorithm has the steps to update g-values of nodes on *CLOSED* as well.
- The reason why A^* may not have found the optimal cost to a node is that it picks nodes based on f-values which have an h-value that could be inaccurate in its estimates.
- The condition that $h(n)$ underestimate the distance to the goal is sufficient for admissibility, and it will find an optimal path to the goal. To find an optimal path to every node *en route*, it needs a stricter condition.

The Monotone Condition

- The monotone or consistency condition is

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

  for all nodes m and its successor n.
- Let n be a successor of m on an optimal path to the goal.
- Remember that the heuristic function is an estimate of the distance to the goal. As one moves towards the goal, the h-value is expected to decrease.
- The monotone condition says that this decline cannot be greater than the edge cost.
- One can think of $h(m) - h(n)$ as the cost of the $m - n$ edge as estimated by the heuristic function. This too must be an underestimate.
- Rearranging the above inequality, we get

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

The Monotone Condition

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

  Adding $g^*(m)$ to both sides,

  $$h(m) + g^*(m) \leq h(n) + k(n, m) + g^*(m)$$
  $$\text{or, } h(m) + g^*(m) \leq h(n) + g^*(n)$$

- A more significant consequence of the monotone condition is that when A^* picks a node n from *OPEN*, it has already found the optimal path from the start node S to n. That is $g(n) = g^*(n)$.

The Monotone Condition

Lemma - L7

Suppose h is an underestimate heuristic function which satisfies the monotone condition. Then, when A^* picks a node n from *OPEN*, it has already found the optimal path from the start node S to n. That is $g(n) = g^*(n)$.

- Let A^* be about to pick node n with a value $g(n)$.
- Let there be an optimal path P from S to n which is yet unexplored fully.
- On this path P let n_L be the last node on *CLOSED* and let n_{L+1} be its successor on *OPEN*. Given the monotone condition,

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

becuase both are on the optimal path.

The Monotone Condition

- By transitivity, the last inequality extends to node n.

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

- Algorithm A^* is about to pick node n when n_{L+1} is on *OPEN*. Hence,

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

- This gives us

$$h(n) + g(n) \leq h(n) + g^*(n)$$
$$\text{or } g(n) \leq g^*(n)$$

- But $g(n)$ cannot be less than $g^*(n)$ which is optimal cost from S to n. Therefore,

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

The Monotone Condition

- A direct consequence of this is that A^* does not have to update $g(n)$ for nodes of *CLOSED*.
- This allows us to implement some space saving versions of A^* which prune *CLOSED* drastically.

Performance of A^*

1. Completeness: The algorithm is complete. As shown earlier, it will even terminate on infinite graphs if there is a path from the start node to the goal.
2. Quality of solution: As shown earlier, A^* is admissible. Given an underestimating heuristic function, it always finds the optimal path to the goal node.

*Performance of A**

3. Space complexity: The space required depends upon how good the heuristic function is. Given a perfect heuristic function, the algorithm heads straight for the goal, and the *OPEN* list will grow linearly. However, in practice, perfect heuristic functions are hard to come by, and it has been experimentally observed that *OPEN* tends to grow exponentially.
4. Time complexity: The time complexity is also dependent on how good the heuristic function is. With a perfect function, it would be linear. In practice, however, it does a fair bit of exploration, reflected by the size of *CLOSED*, and generally needs exponential time as well.

Some Variants of A^*

Weighted A^*

*Weighted A**

- We have observed that A^* explores more of the space than BestFirstSearch, but finds an optimal solution.
- We look at a variation of A^* that allows us to choose the trade-off between quality and complexity, of both time and space which go hand in hand for A^*.
- We have also observed earlier that heuristic functions with higher estimates result in more focussed search.
- We have so far imposed a condition that the heuristic function must underestimate the distance to the goal.
- We now explore how the algorithm behaves when we relax that condition.

*Weighted A**

- Consider a weighted version of the estimated cost

$$f(n) = \alpha g(n) + \beta h(n).$$

  What would be the behaviour of A^* with different values of α and β?
- When $\beta = 0$, we have the uninformed search algorithms BFS and $\mathrm{B\&B}$.
- We can model DFS by defining $g(n) = 1/\text{depth}$.
- If $\alpha = 0$, we have the $\textsc{BestFirstSearch}$.

*Weighted A**

Consider

$$f(n) = \alpha g(n) + \beta h(n).$$

- When $\alpha = 1$ and $\beta = 1$, we have A^*.
- Choosing a value of $\alpha = 1$ and $\beta < 1$ would push the algorithm towards B&B.
- Choosing $\beta > 1$ gives us the weighted A^* algorithm.
- Traditionally in the literature, the algorithm is known as wA^* where w is the weight in

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

- As the value of w increases, the algorithm becomes more and more like BestFirstSearch, finding the solution faster but possibly one with a higher cost.

Space Saving Versions of A^*

Iterative Deepening A^*

IDA*

- Algorithm iterative deepening A^* (*IDA**) replicates the strategy used by the algorithm DFID, which is to mimic a BFS using a series of DFSs of increasing depth.
- *IDA** is to A^* as DFID is to BFS, finding the optimal path but using linear space.
- The main difference is that edges have a **non-uniform cost**, and the **depth of a node is not a measure of the cost**.
- Instead we use the f-values to determine the bound till which DFS searches in each cycle.

IDA*

IDA* does a series of DB-DFSs with increasing depth bounds.

IDA*(S)
  depthBound $\leftarrow f(S)$
  while True
    DB-DFS(*S*, *depthBound*)
    depthBound $\leftarrow f(N)$ of cheapest unexpanded node on *OPEN*

IDA*

- The above algorithm suffers from the same drawback we observed in DFID. The algorithm may loop forever even for a finite graph which does not contain the goal node.
- We can modify *IDA** to count the number of new nodes generated by it and terminate with failure, like we did for DFID, if the count does not increase in some cycle.
- If the graph is infinite and there is no goal node, there is no way of reporting failure.

Thank You