Artificial Intelligence:
Search Methods for Problem Solving

State Space Search

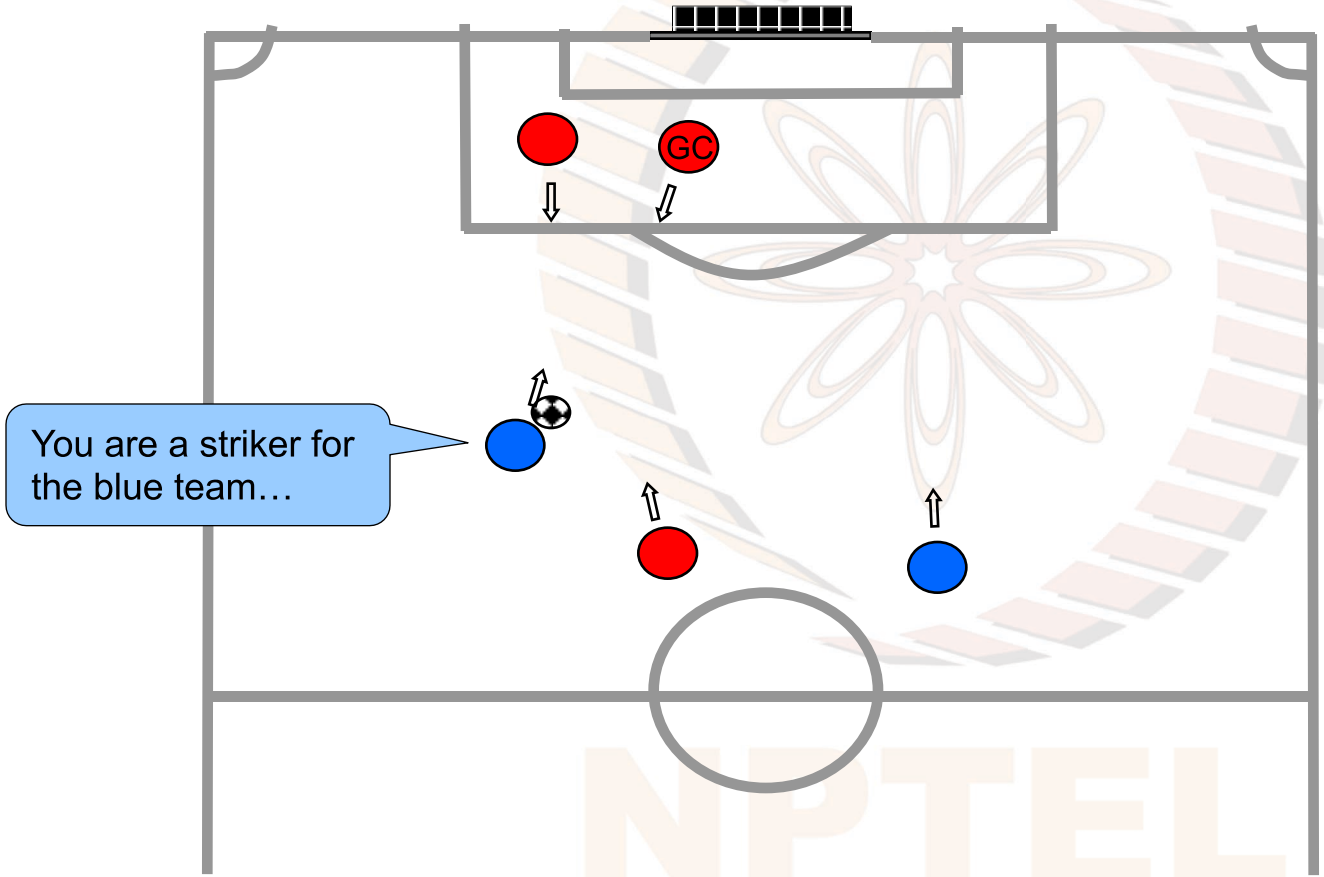
A First Course in Artificial Intelligence: Chapter 2

Deepak Khemani

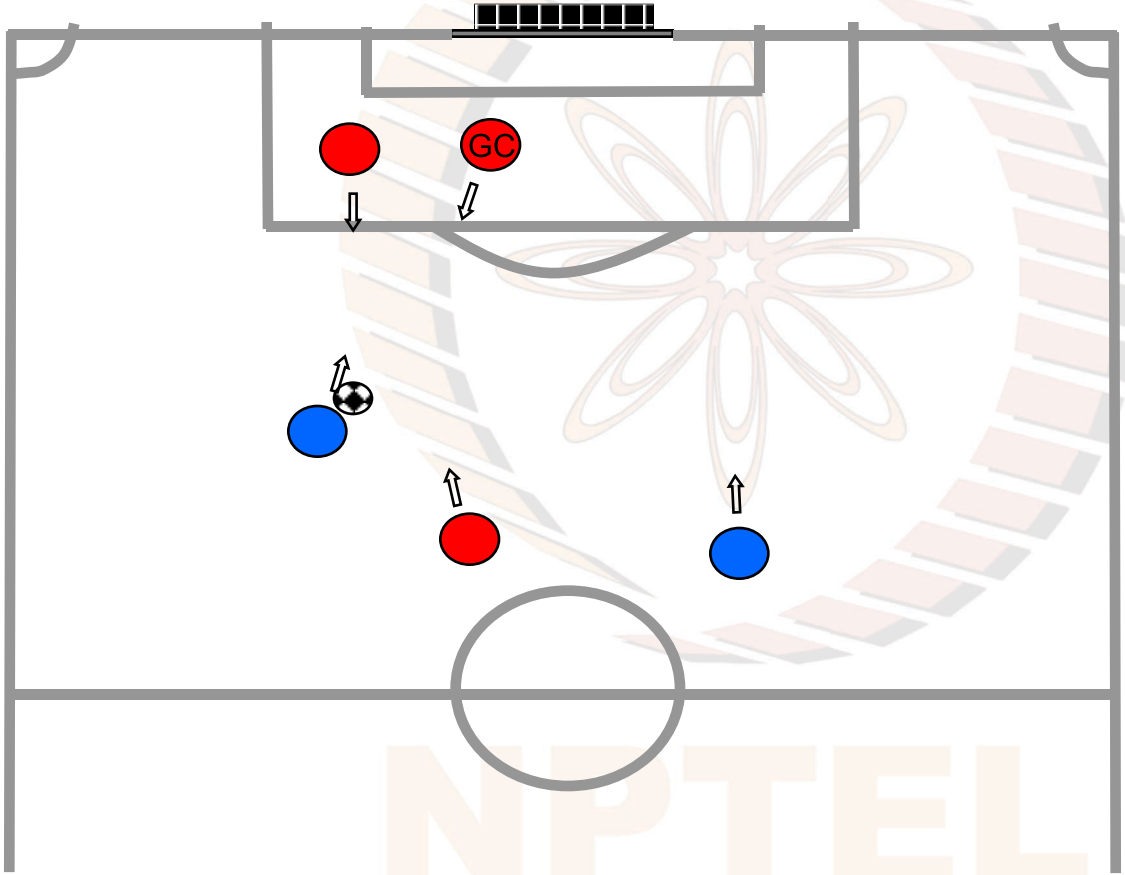
Department of Computer Science & Engineering

IIT Madras

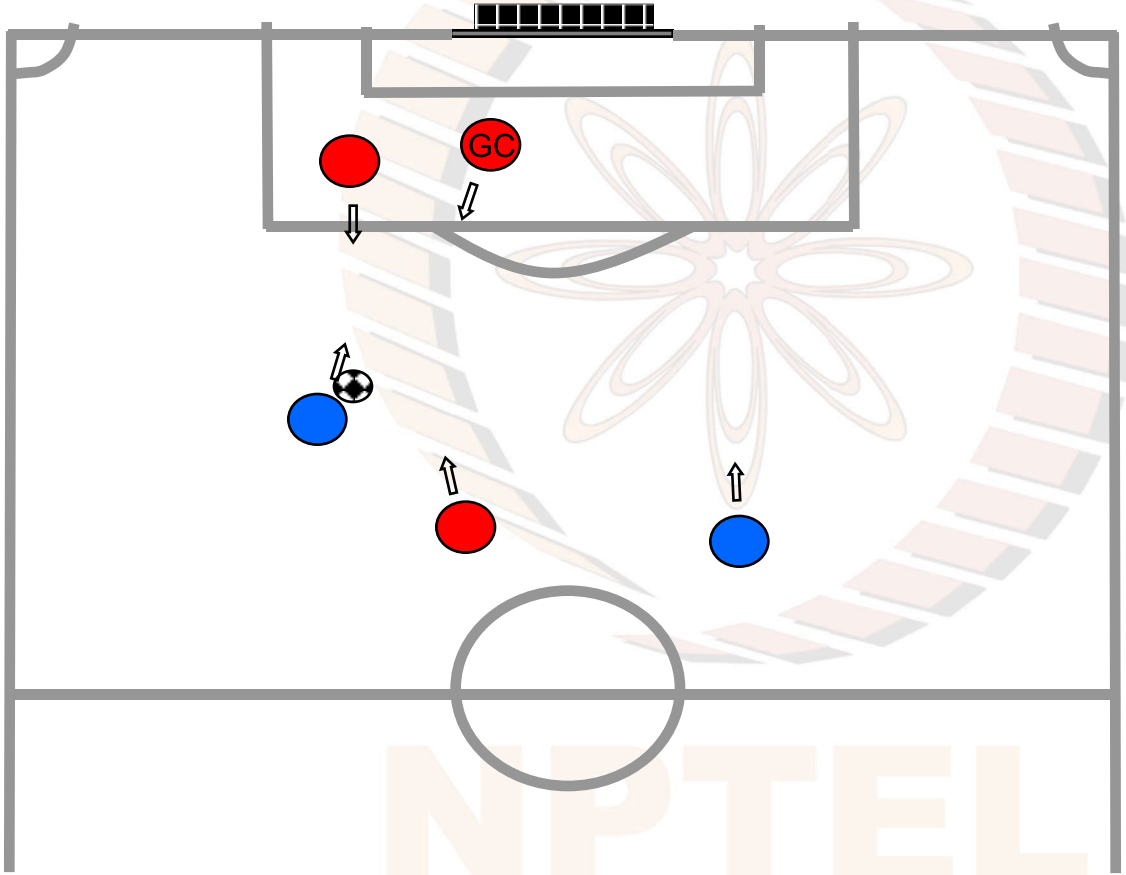
On a football field…



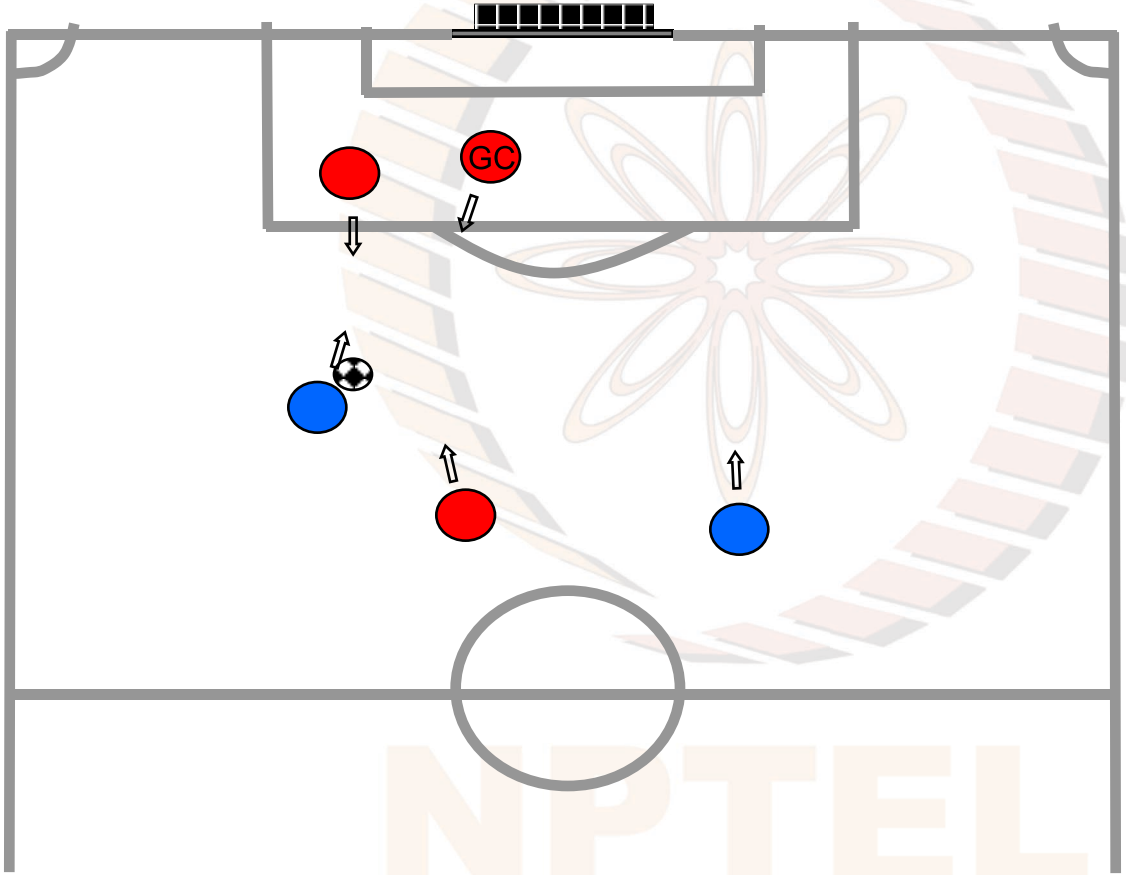
.. moving forward with the ball…



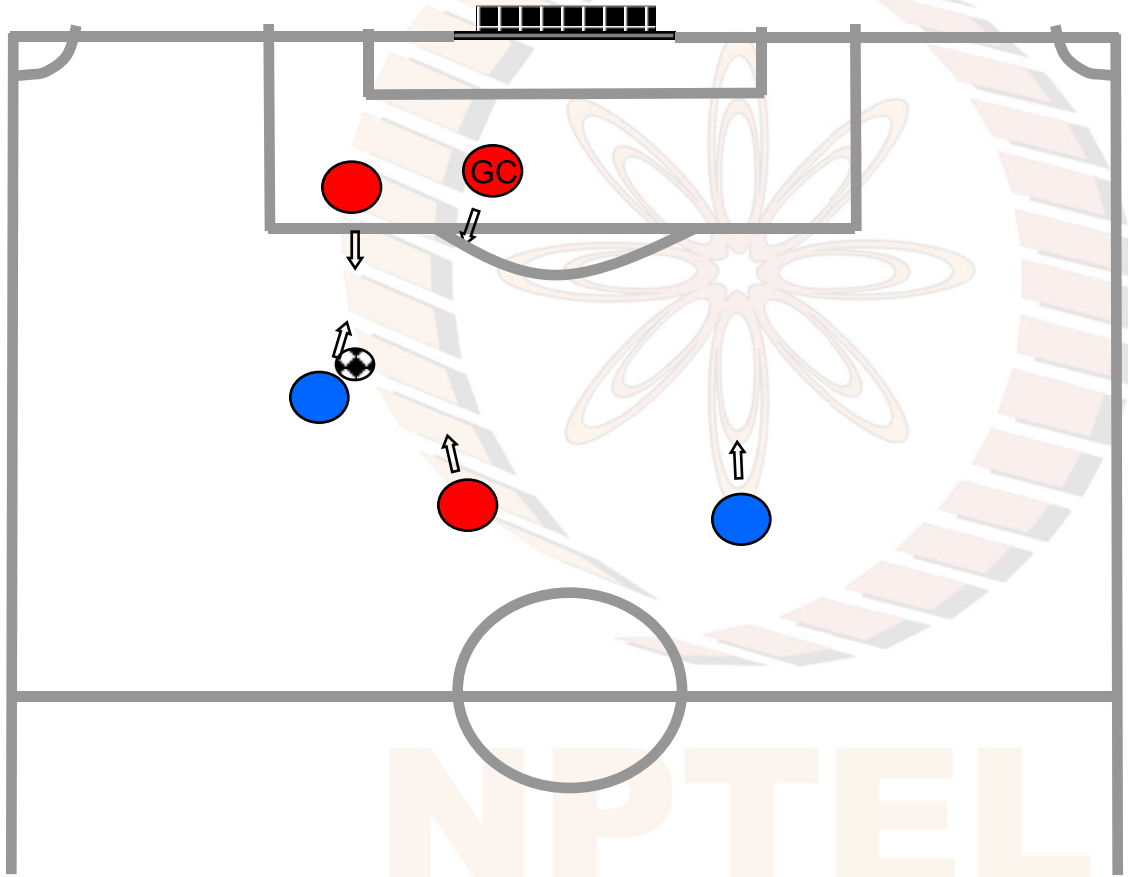
.. moving forward with the ball…



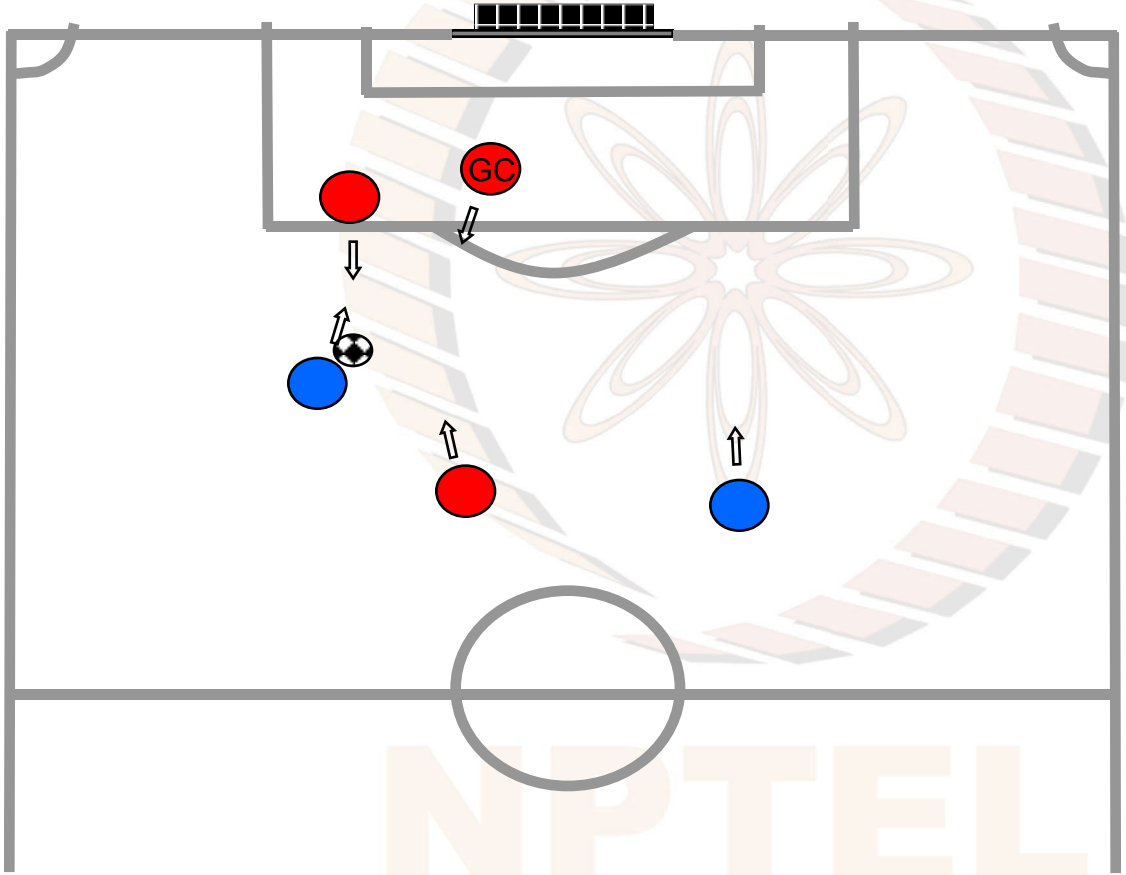
.. moving forward with the ball…



… the defenders are closing in …



… the defenders are closing in … what should you do?



Problem Solving

An autonomous agent

in some world

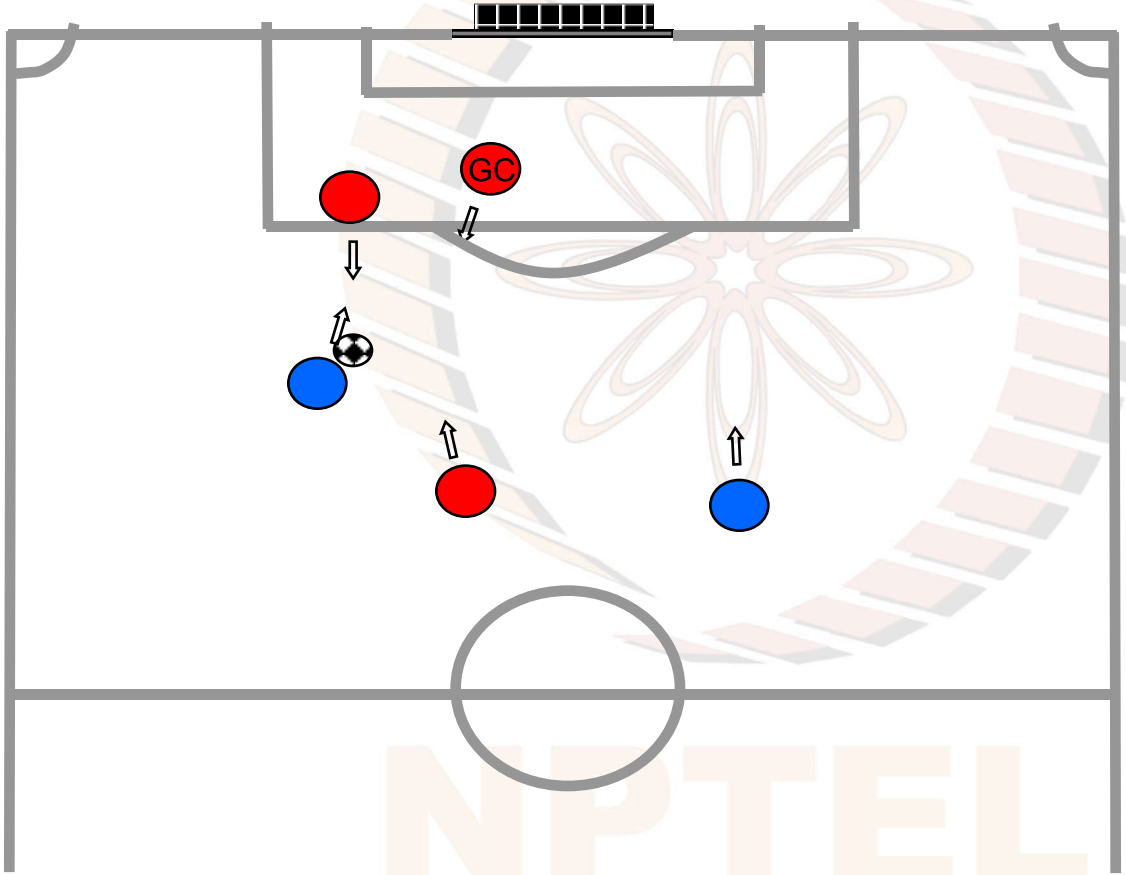
has a goal to achieve

and

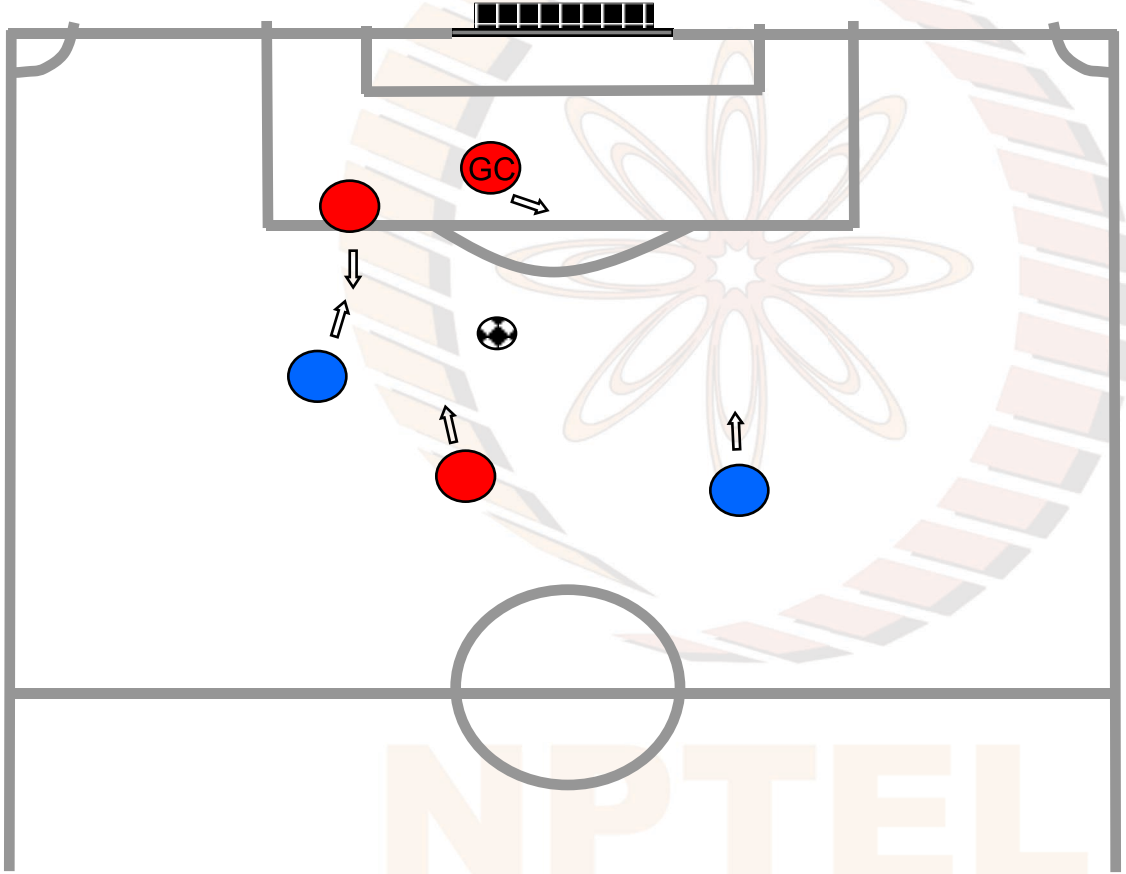
a set of actions to choose from

to strive for the goal

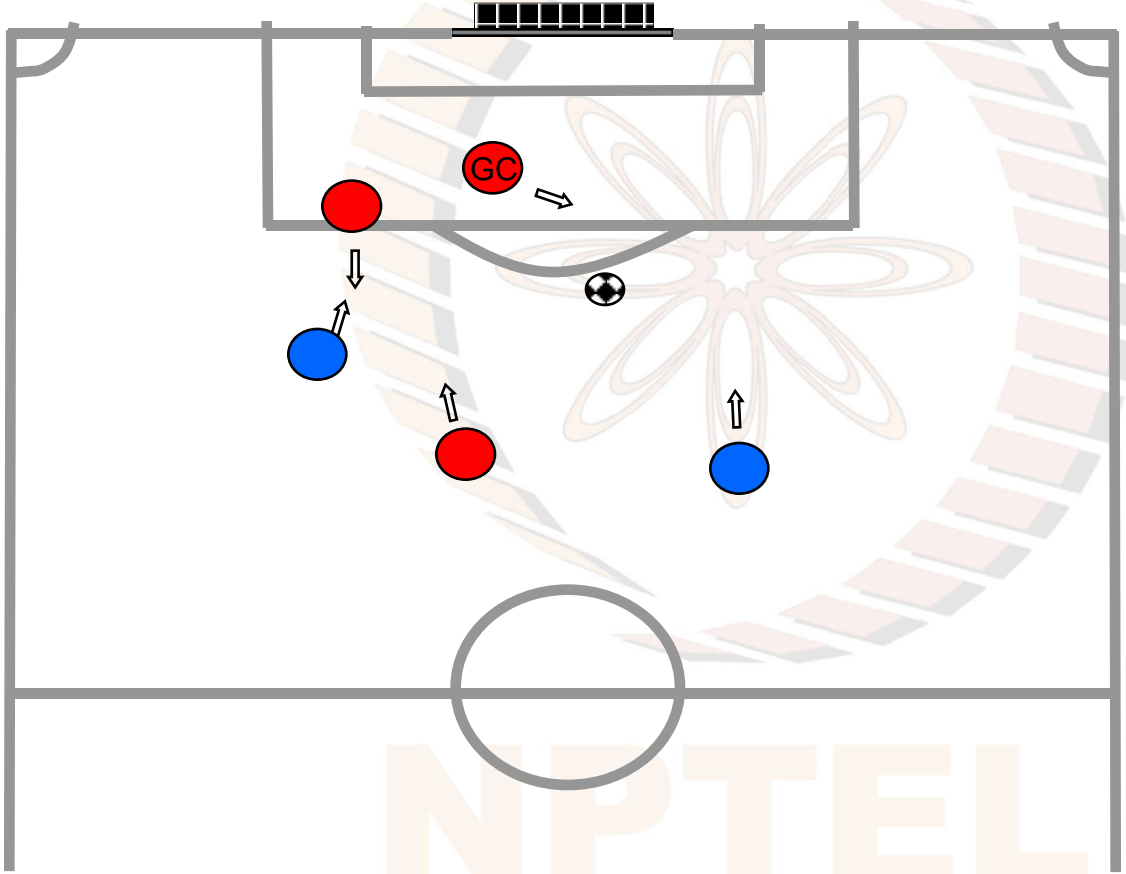
An intelligent agent would…



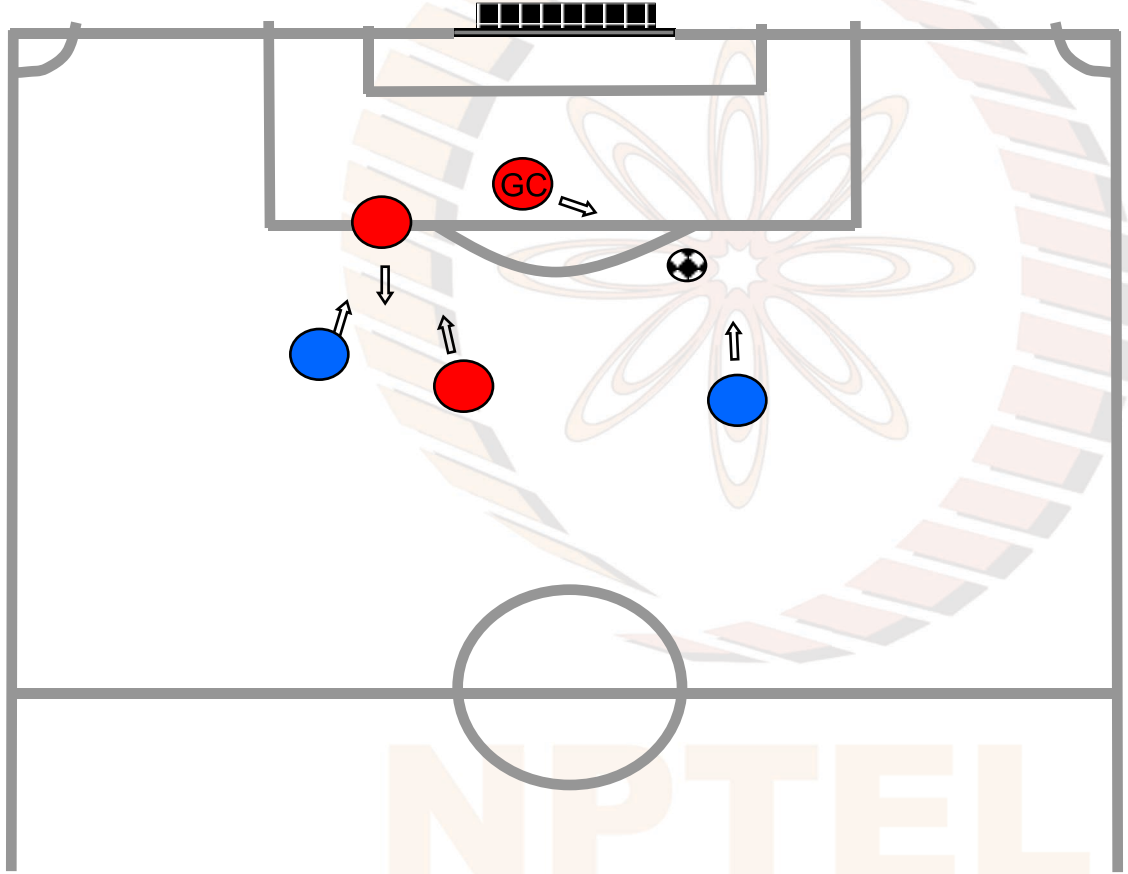
Pass the ball to a teammate who is free…



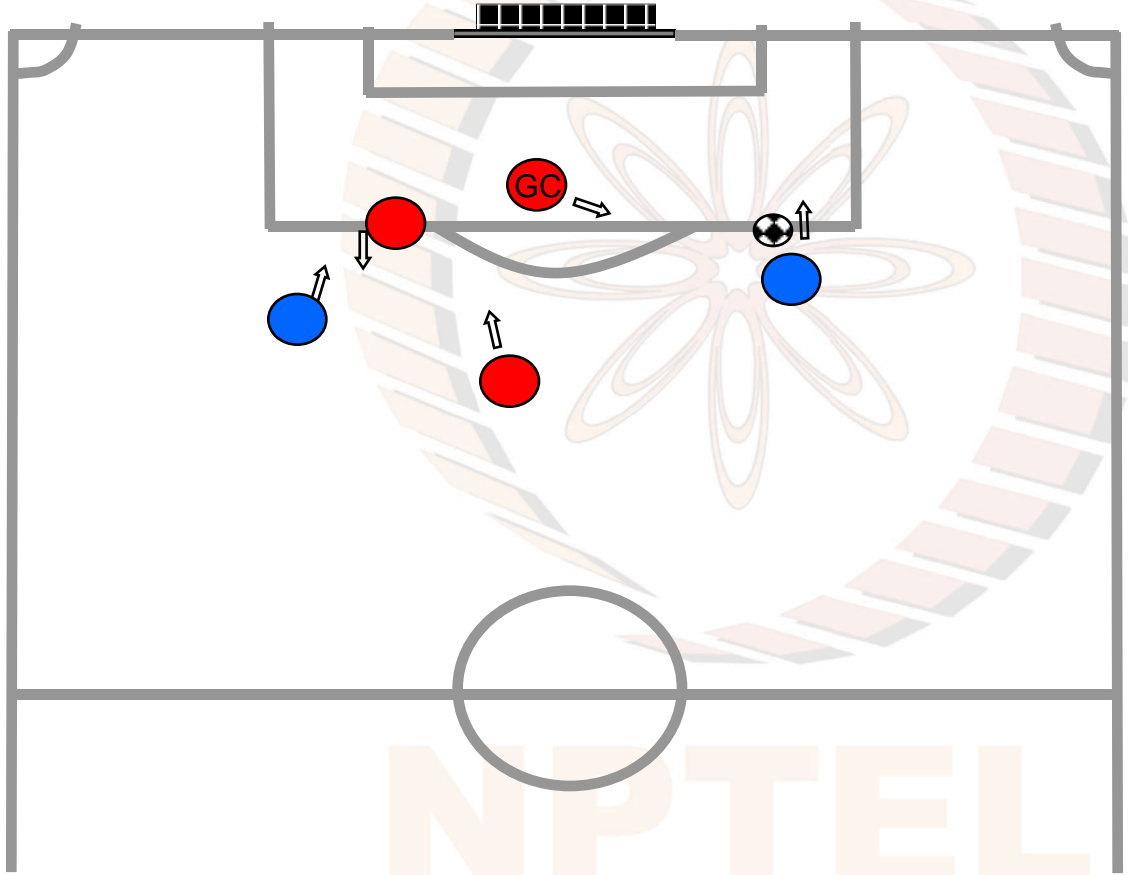
Pass the ball to a teammate who is free…



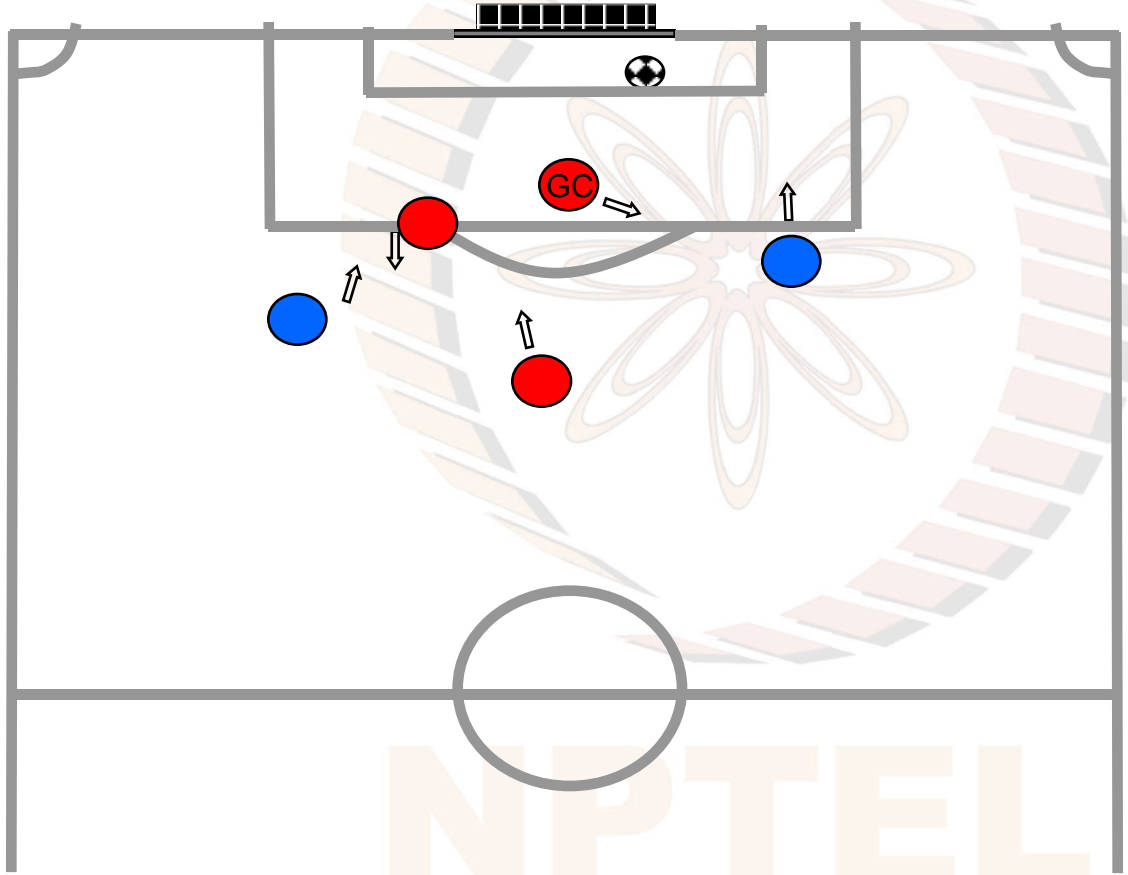
Pass the ball to a teammate who is free…



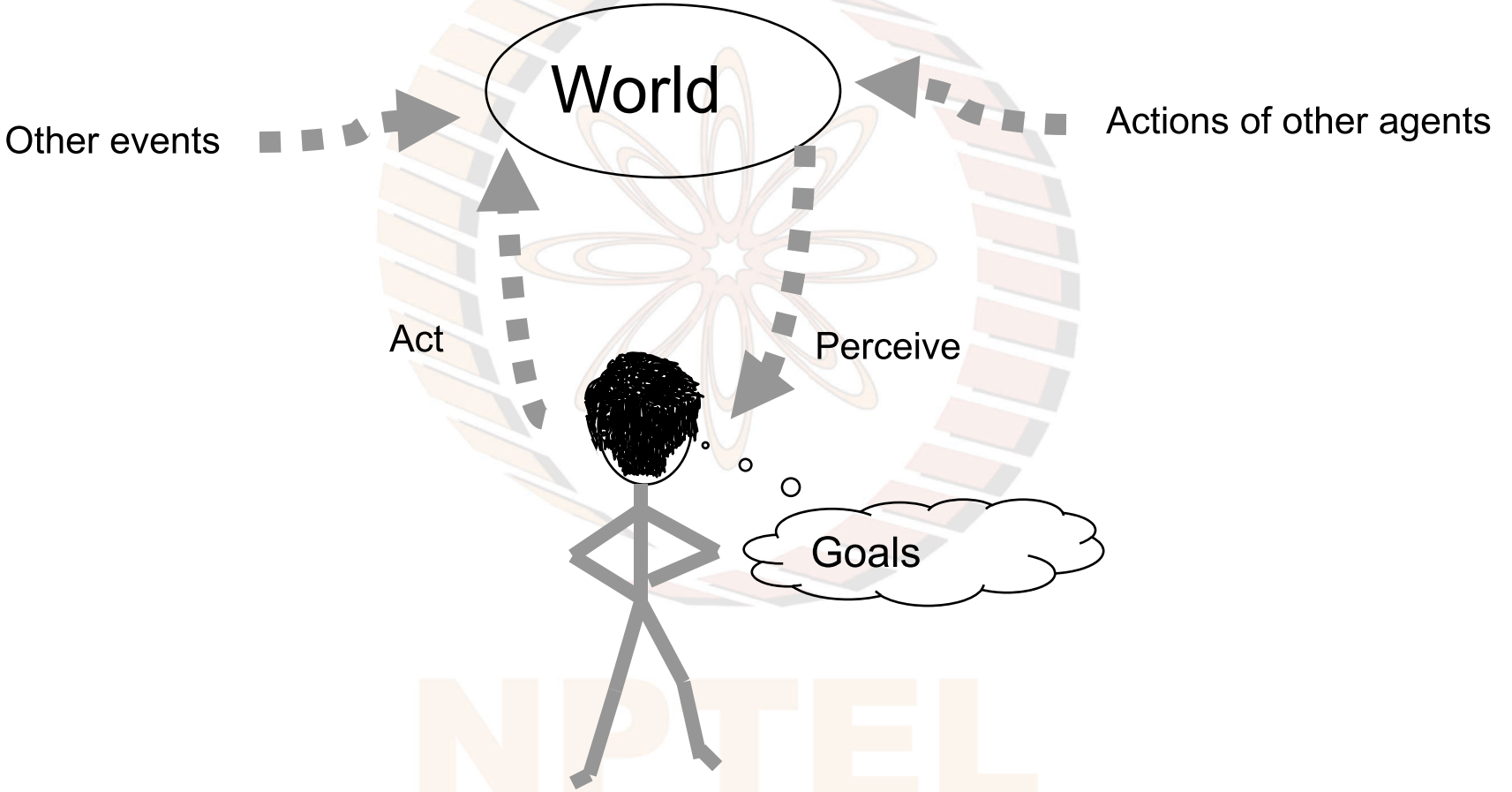
… and has a better chance to score



… and has a better chance to score



The real world is complex!



Must learn to walk before one can run

Deal with simple problems first

- The world is static
- The world is completely known
- Only one agent changes the world
- Actions never fail
- Representation of the world is taken care of (to start with at least)

Past vs. Future

There are two main approaches to problem solving.

Exploit experience

A memory based agent looks into the past

Knowledge essentially links the past to the present.
– The motto is to avoid reinventing the wheel.

The danger : “You never step into the same river twice”
– the problem rarely repeats itself exactly.

Use first principles

A trial and error simulation peers into the future.

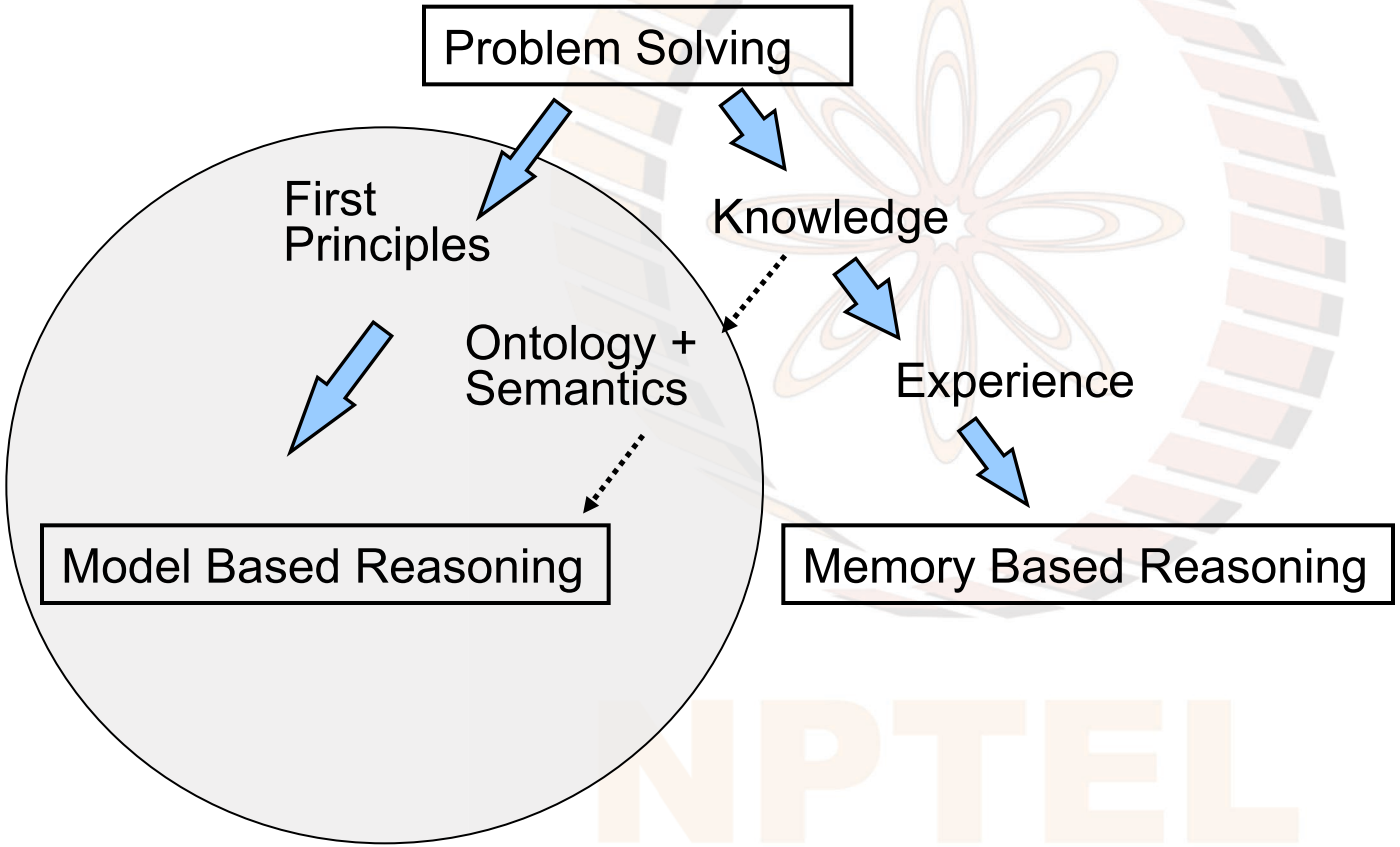
The solution is always for the problem being solved.

Drawbacks: Simulation often requires sophisticated modelling.

Also the “the wheel is reinvented again and again”.

Problem Solving

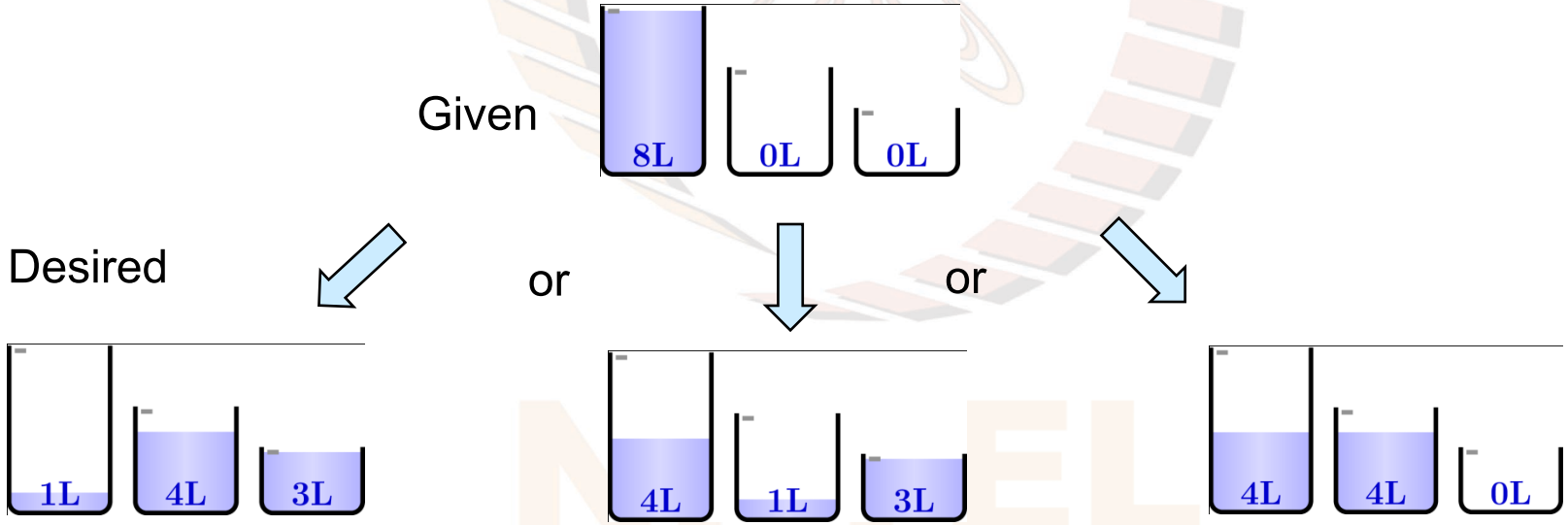
Humankind is a problem solving species.



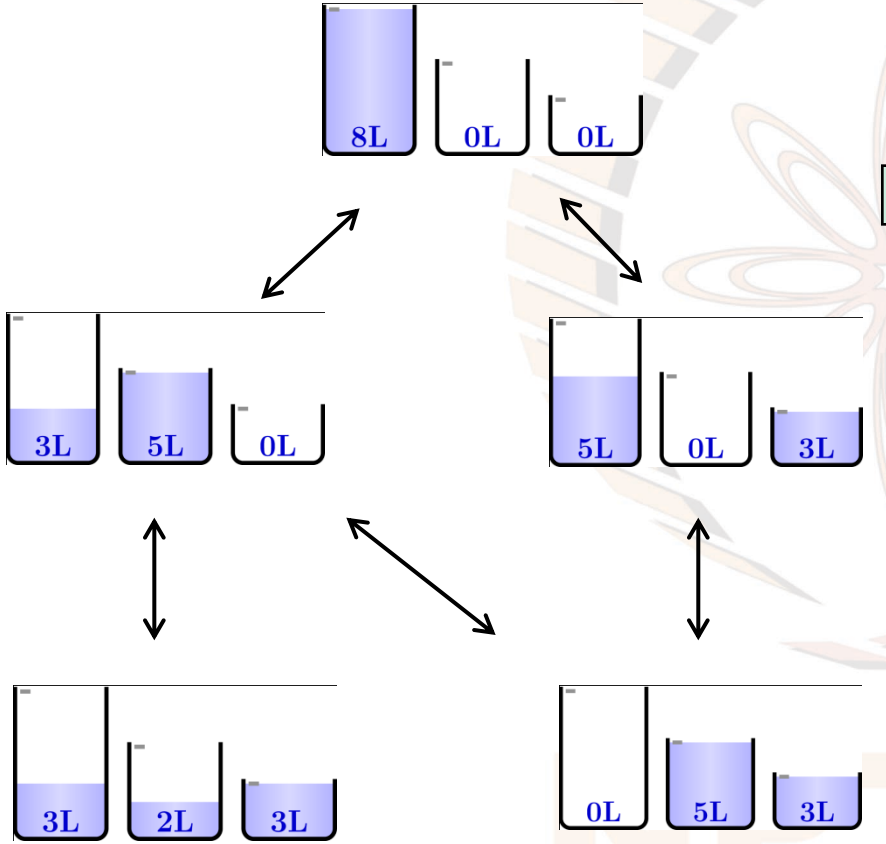
Some sample problems

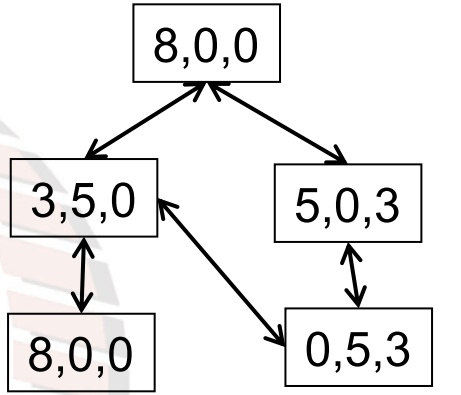
The Water Jug problem

- You have three jugs with capacity 8, 5 and 3 liters
- The 8-liter jug is filled with water.
- The 5 liter and 3 liter jugs are empty.
- You are required to measure (say) 4 liters of water



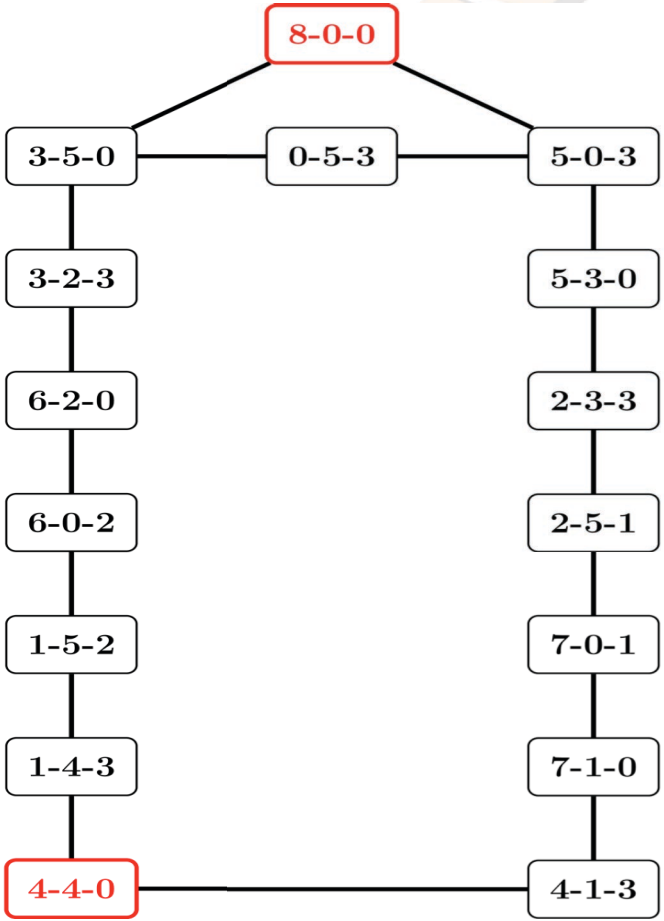
Water Jug Problem : Some Sample Moves





State transition graph

The Solution : (8,0,0) → (4,4,0)



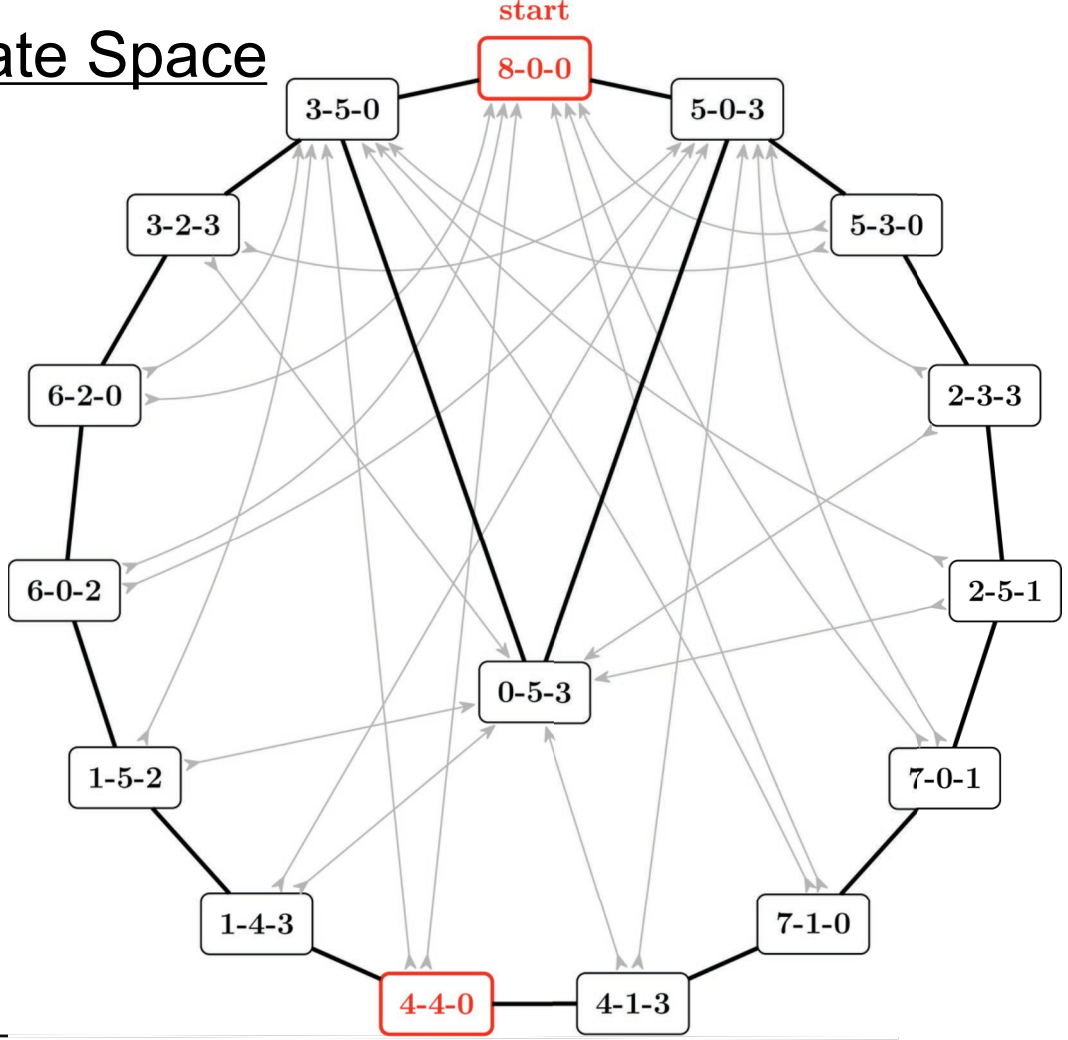
Various solutions for measuring 4 liters exist

We are not consider those cases where one can throw away some water.

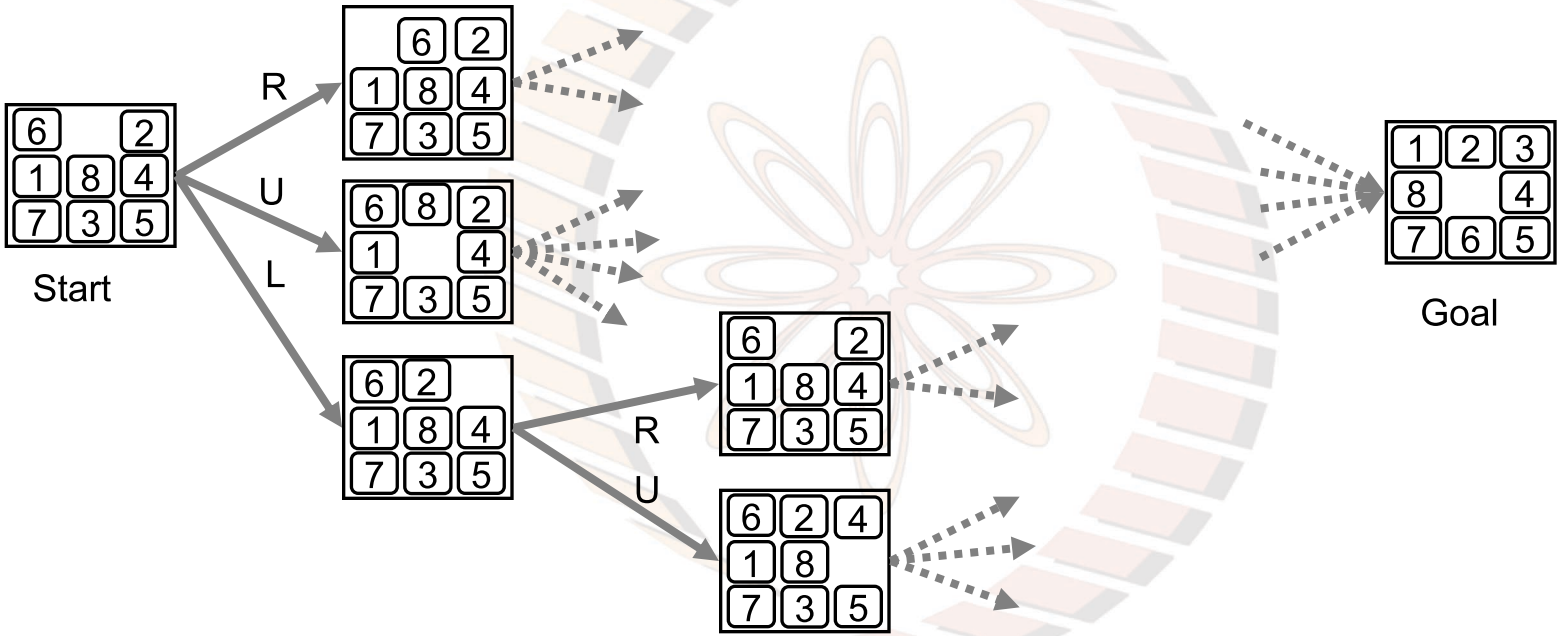
The Complete State Space

Observe that some moves are reversible.
For example,
(8,0,0) ⟷ (3,5,0)

And that some moves are not reversible.
For example,
(3,2,3) ⟶ (0,5,3)



The Eight-puzzle



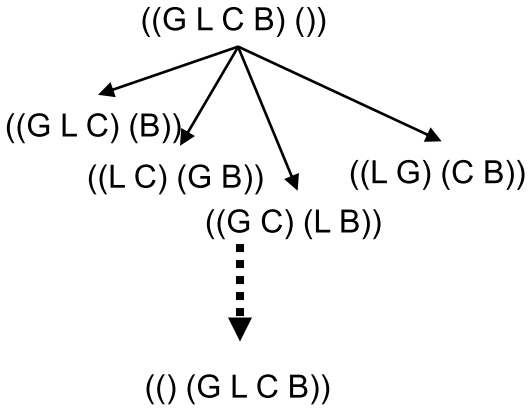
The Eight puzzle consists of eight tiles on a 3x3 grid. A tile can slide into an adjacent location if it is empty. A move is labeled R if a tile moves right, and likewise for up (U), down (D) and left (L).

Man, Goat, Lion, Cabbage

The MGLC Problem

A man needs to transport a lion, a goat, and a cabbage across a river. He has a boat in which he can take only one of them at a time. It is only his presence that prevents the lion from eating the goat, and the goat from eating the cabbage. He cannot leave the goat alone with the lion, nor the cabbage with the goat. How can he take them all across the river?

((Left bank objects) (Right bank objects))



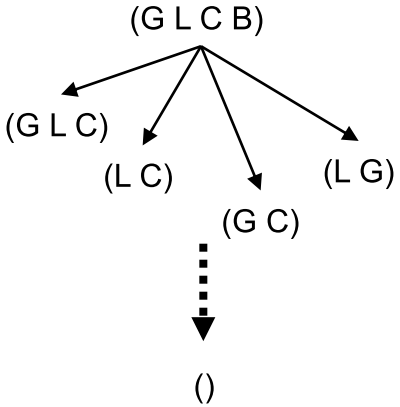
man M is where the boat B is

Variations in Representations

The MGLC Problem

A man needs to transport a lion, a goat, and a cabbage across a river. He has a boat in which he can take only one of them at a time. It is only his presence that prevents the lion from eating the goat, and the goat from eating the cabbage. He cannot leave the goat alone with the lion, nor the cabbage with the goat. How can he take them all across the river?

(Left bank objects)



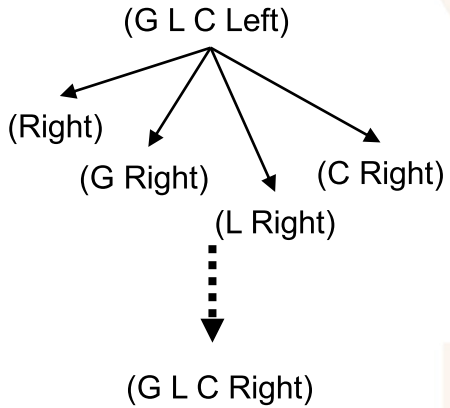
man M is where the boat B is

Which bank is the boat on?

The MGLC Problem

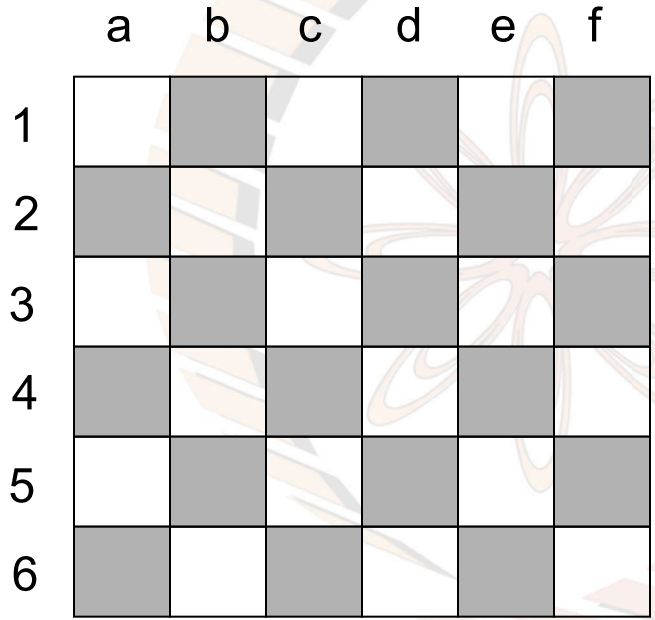
A man needs to transport a lion, a goat, and a cabbage across a river. He has a boat in which he can take only one of them at a time. It is only his presence that prevents the lion from eating the goat, and the goat from eating the cabbage. He cannot leave the goat alone with the lion, nor the cabbage with the goat. How can he take them all across the river?

Objects on the side where the boat is



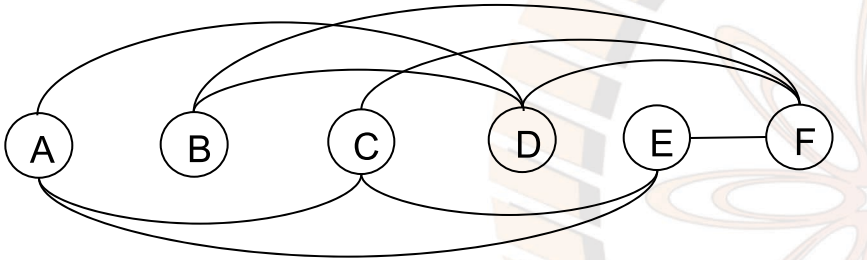
Man M is in each state shown

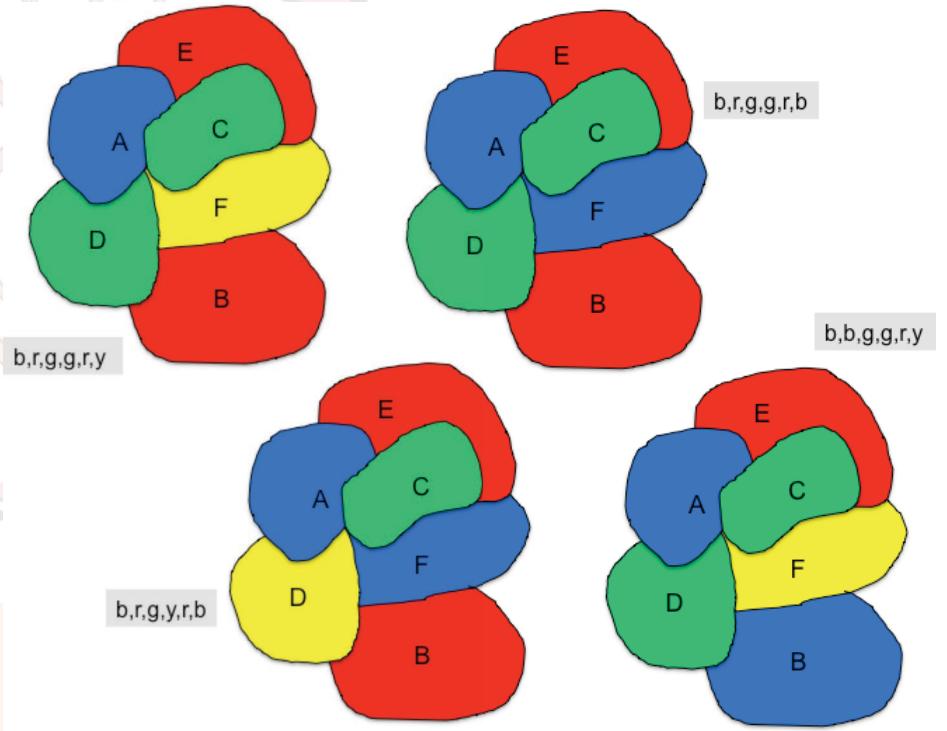
The 6 queens problem



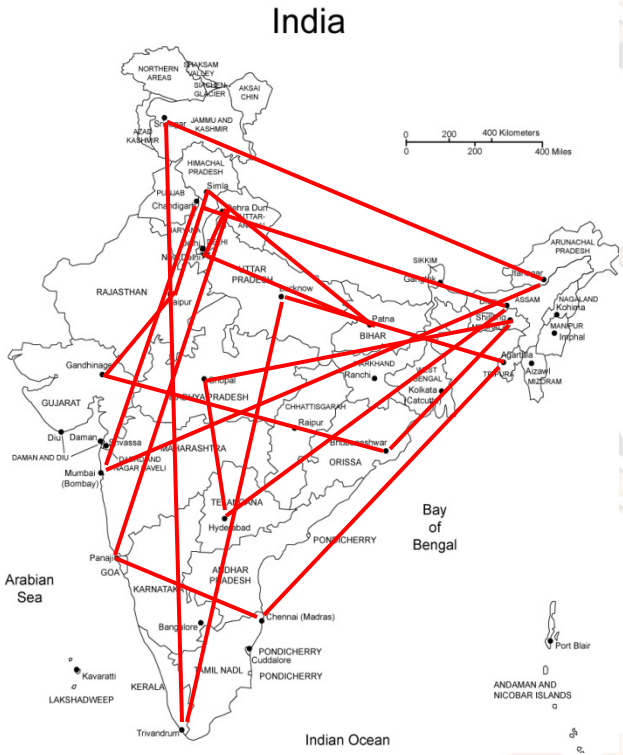
The six queen problem is to place the six queens on a 6x6 chess board such that no queen attacks another.

A map colouring problem and its solutions



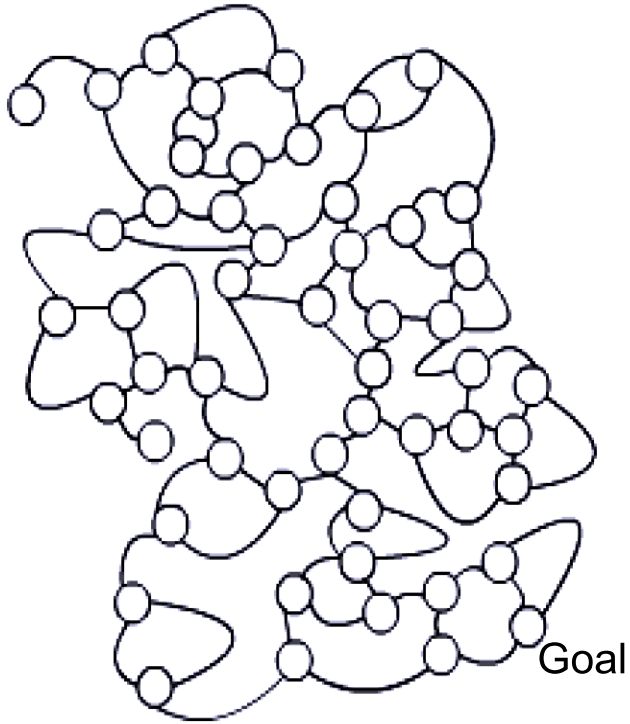


The Traveling Salesman Problem – The Holy Grail of Computer Science

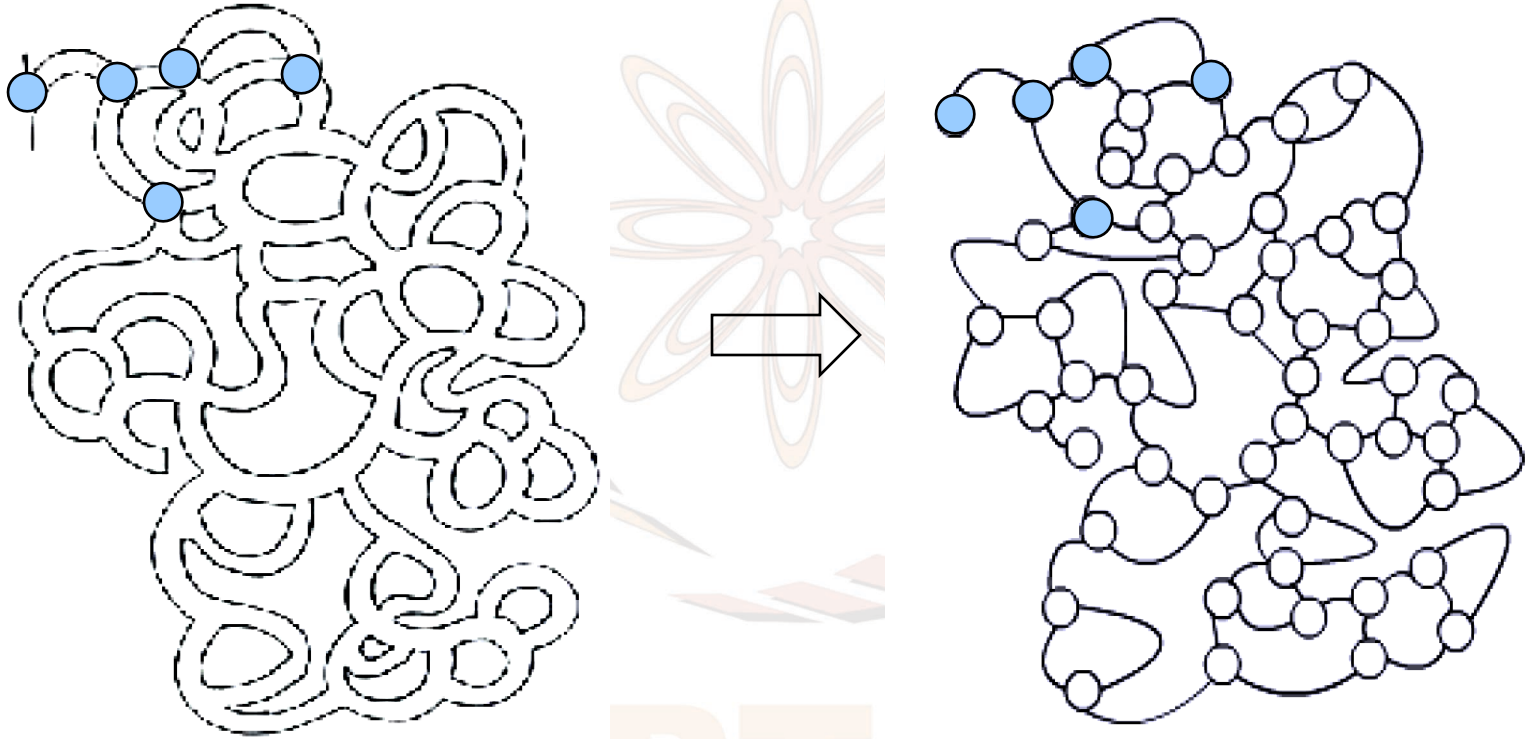


http://www.freeusandworldmaps.com/html/Countries/Asia%20Countries/IndiaPrint.html

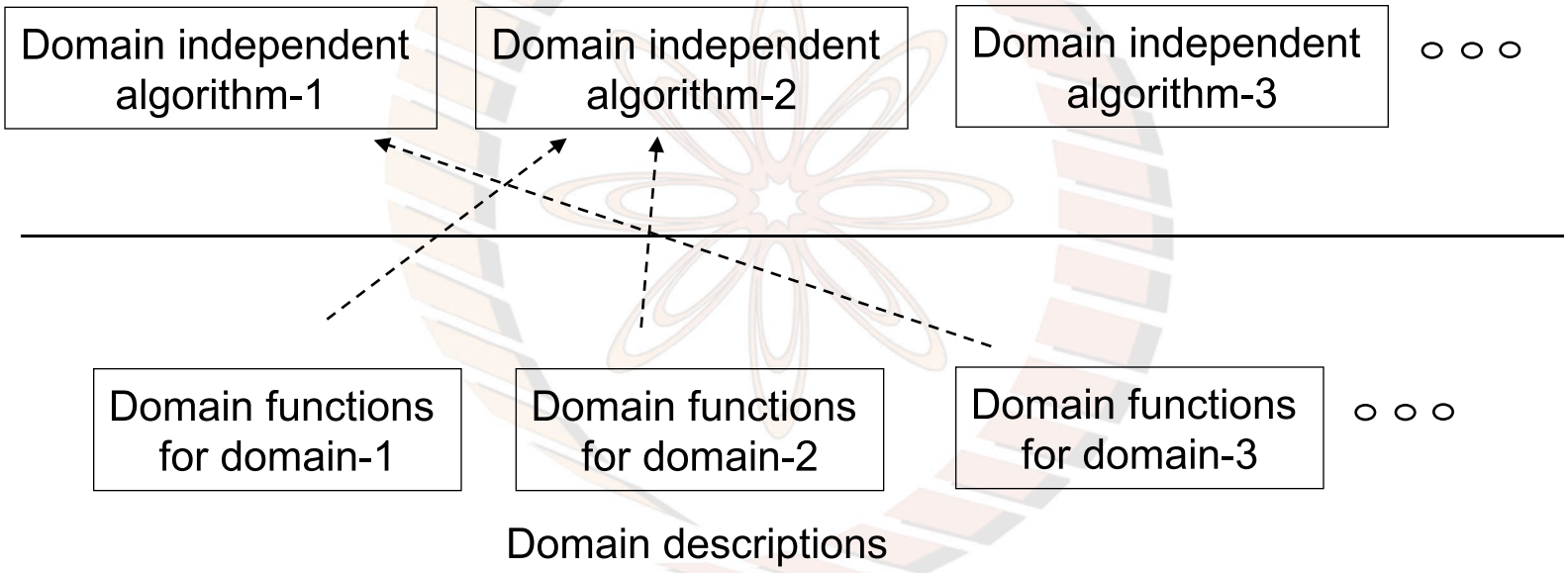
Path finding in a maze – graph search



Every choice point becomes a node in the graph



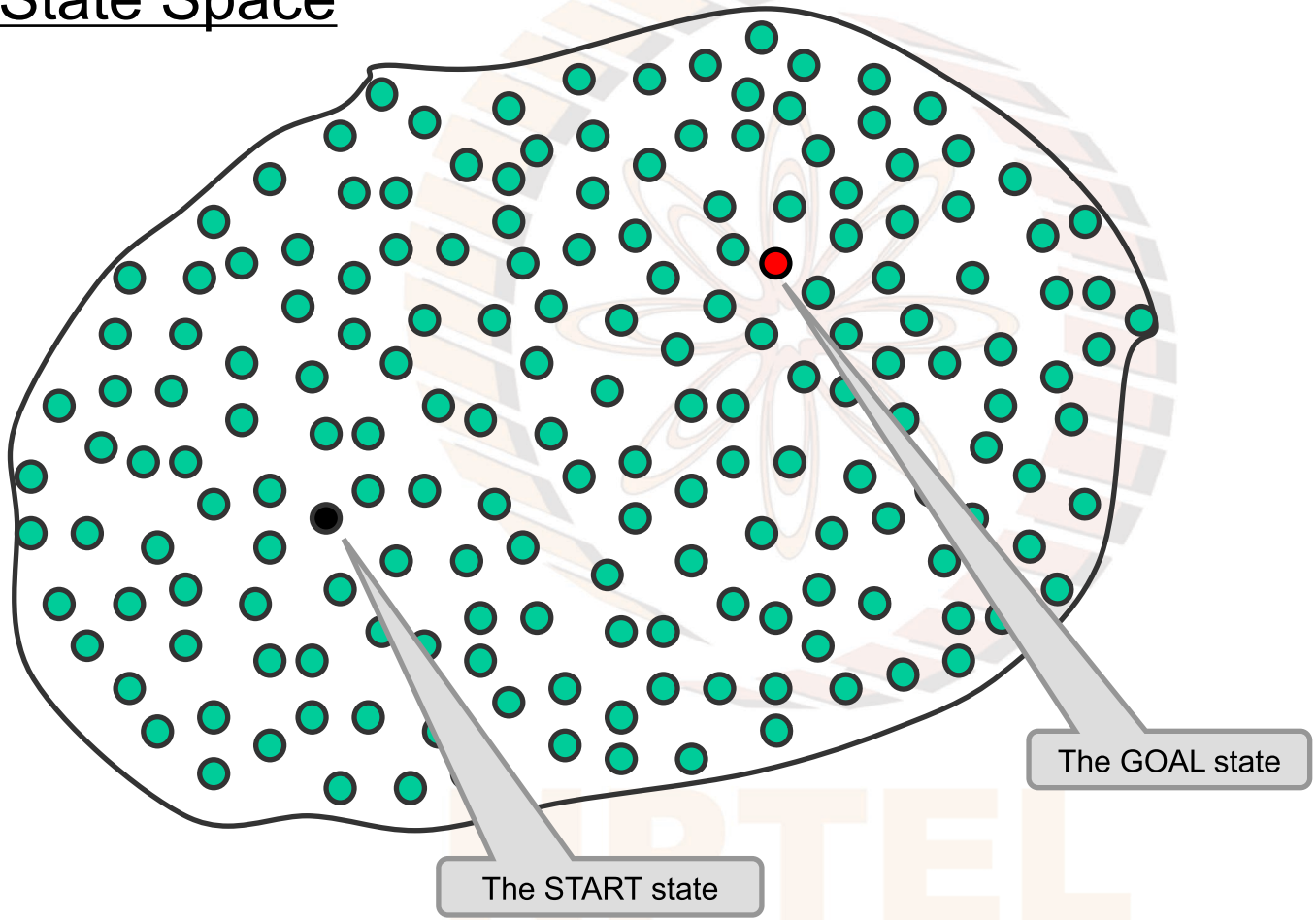
Domain independent problem solving algorithms



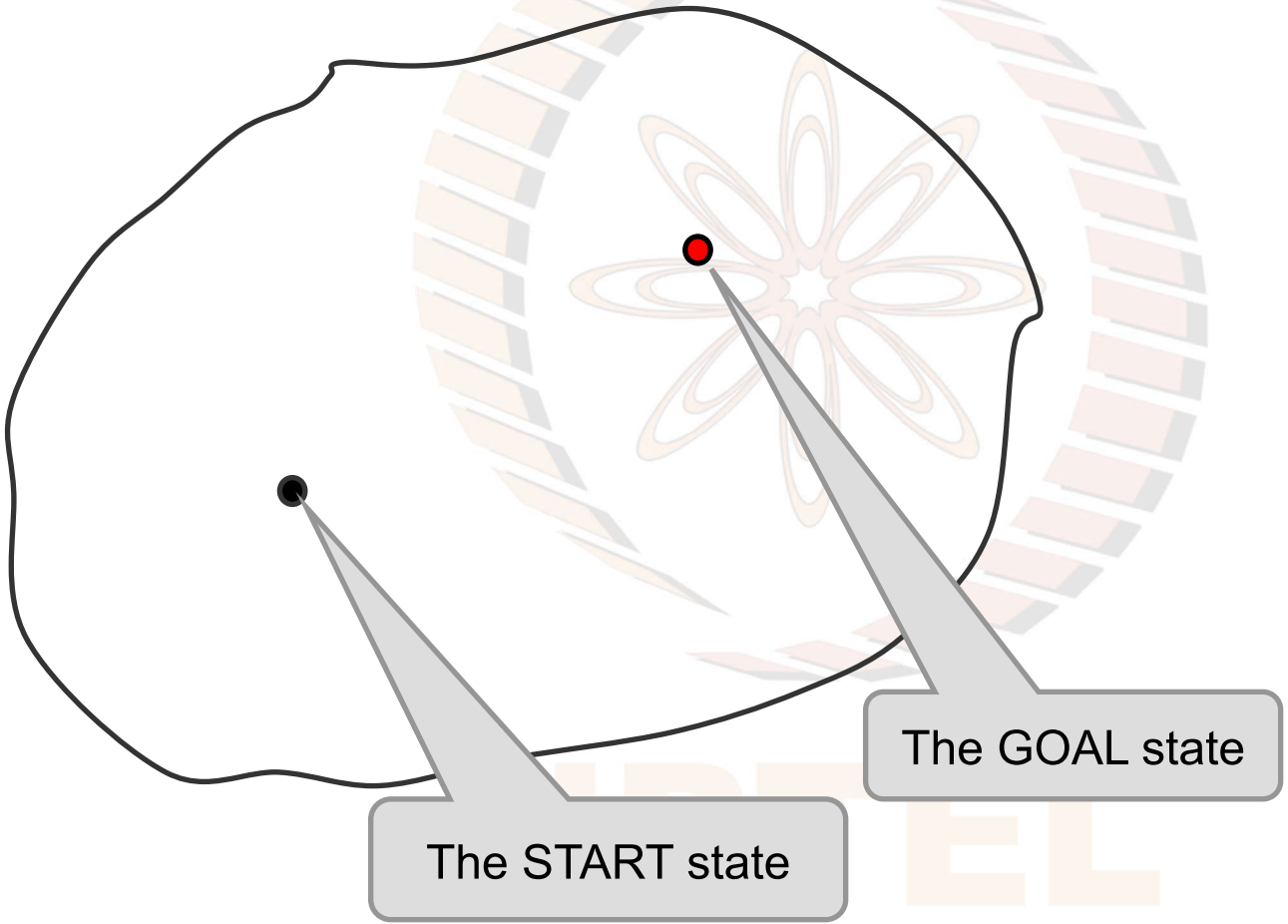
Develop general problem solving algorithms in a domain independent form.

When a problem is to be solved in a domain then the user needs to create a domain description and plug it in a suitable problem solving algorithm.

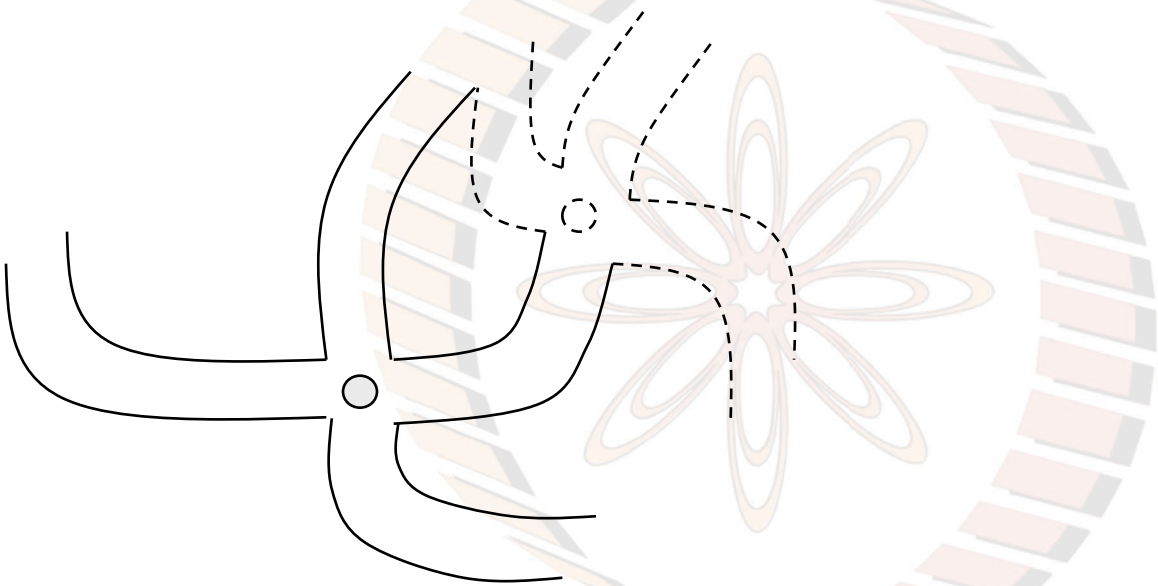
The State Space



The State Space is Implicit



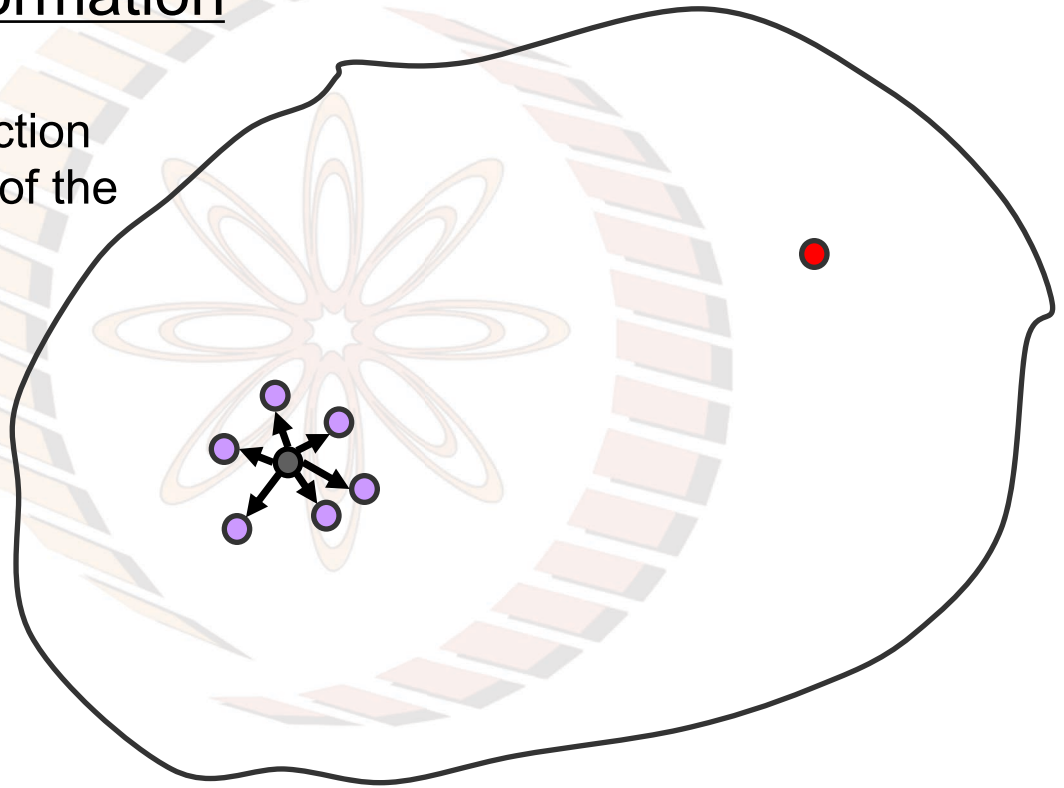
A search node is like a junction in a maze



In a maze one can only see the immediate options. Only when you choose one of them do the further options reveal themselves

Moves : State Transformation

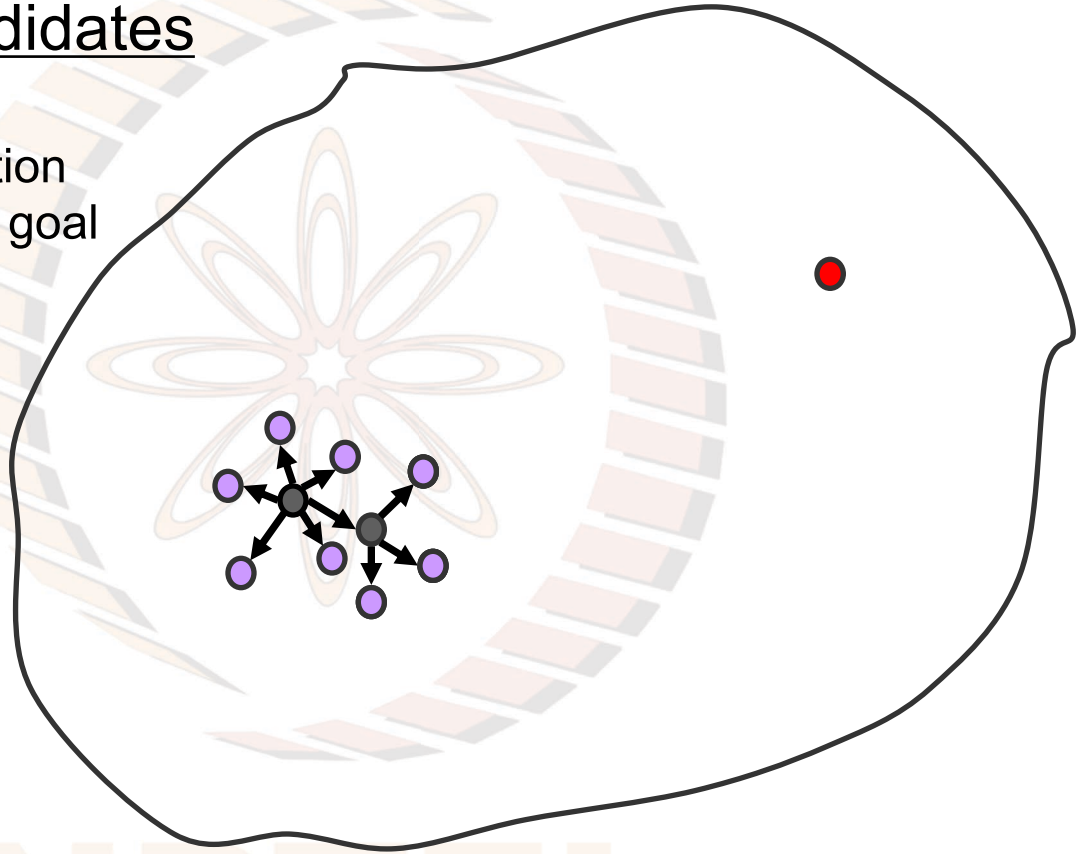
MoveGen(N): A domain function that returns the neighbours of the node N.
-- provided by the user.



A search algorithm has to choose a node from a set of candidates.

The set OPEN of candidates

GoalTest(N): A domain function that tells you whether N is a goal node or not.
-- provided by the user.



OPEN is the set of candidate nodes, shown in purple, generated but not inspected

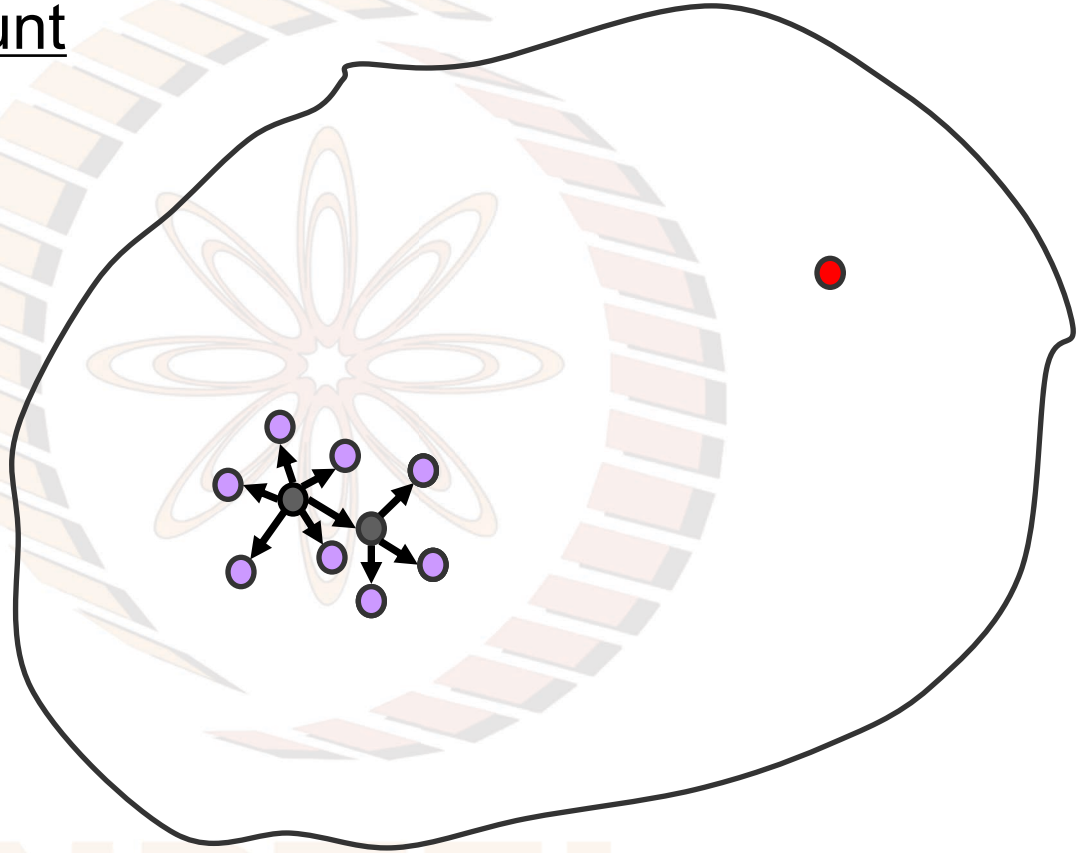
Search = Treasure Hunt

Generate & Test :

Traverse the space by generating new nodes (using MoveGen)

and

test each node (using GoalTest) whether it is the goal or not



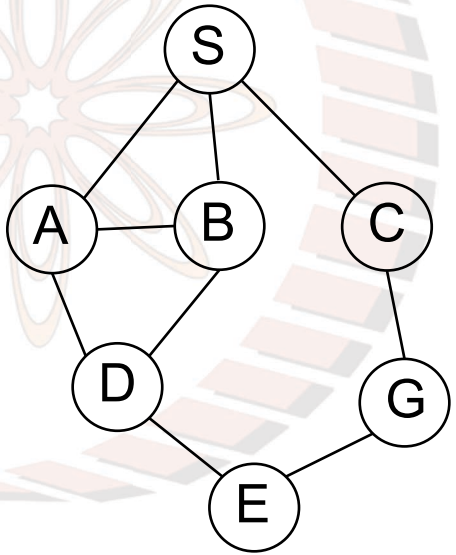
Key question : *Which* node to pick from the set of candidates?!

A Tiny Search Problem

The *MoveGen* function

S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)

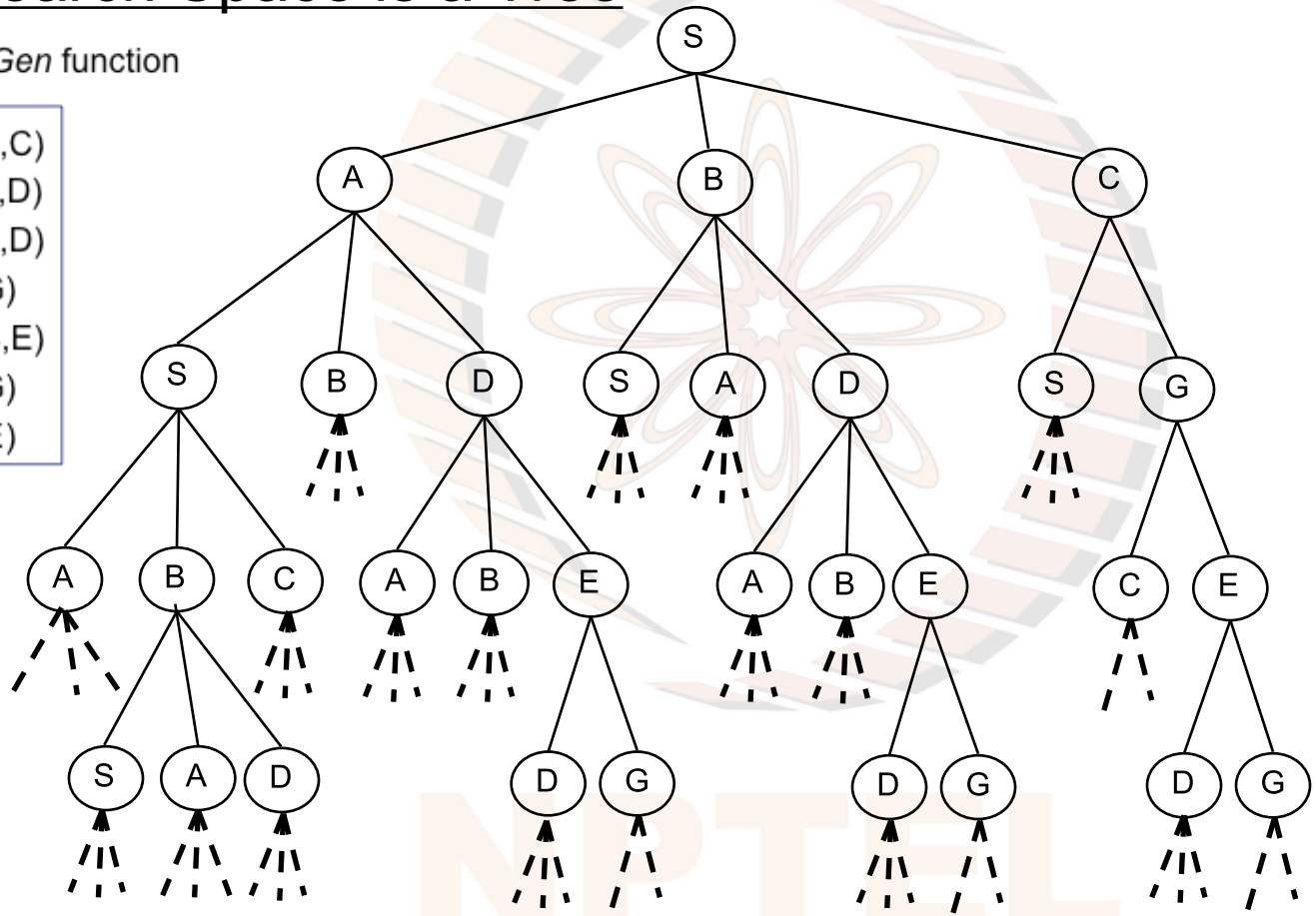
The State Space



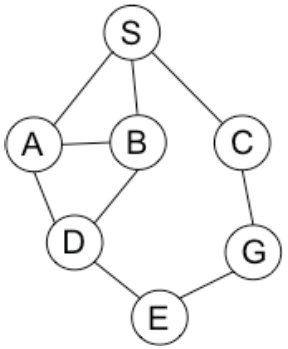
The Search Space is a Tree

The *MoveGen* function

S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



The State Space



Simple Search 1

SimpleSearch1 simply picks a node N from OPEN and checks if it is the goal.
-If Yes it returns N
-If No it adds children of N to OPEN

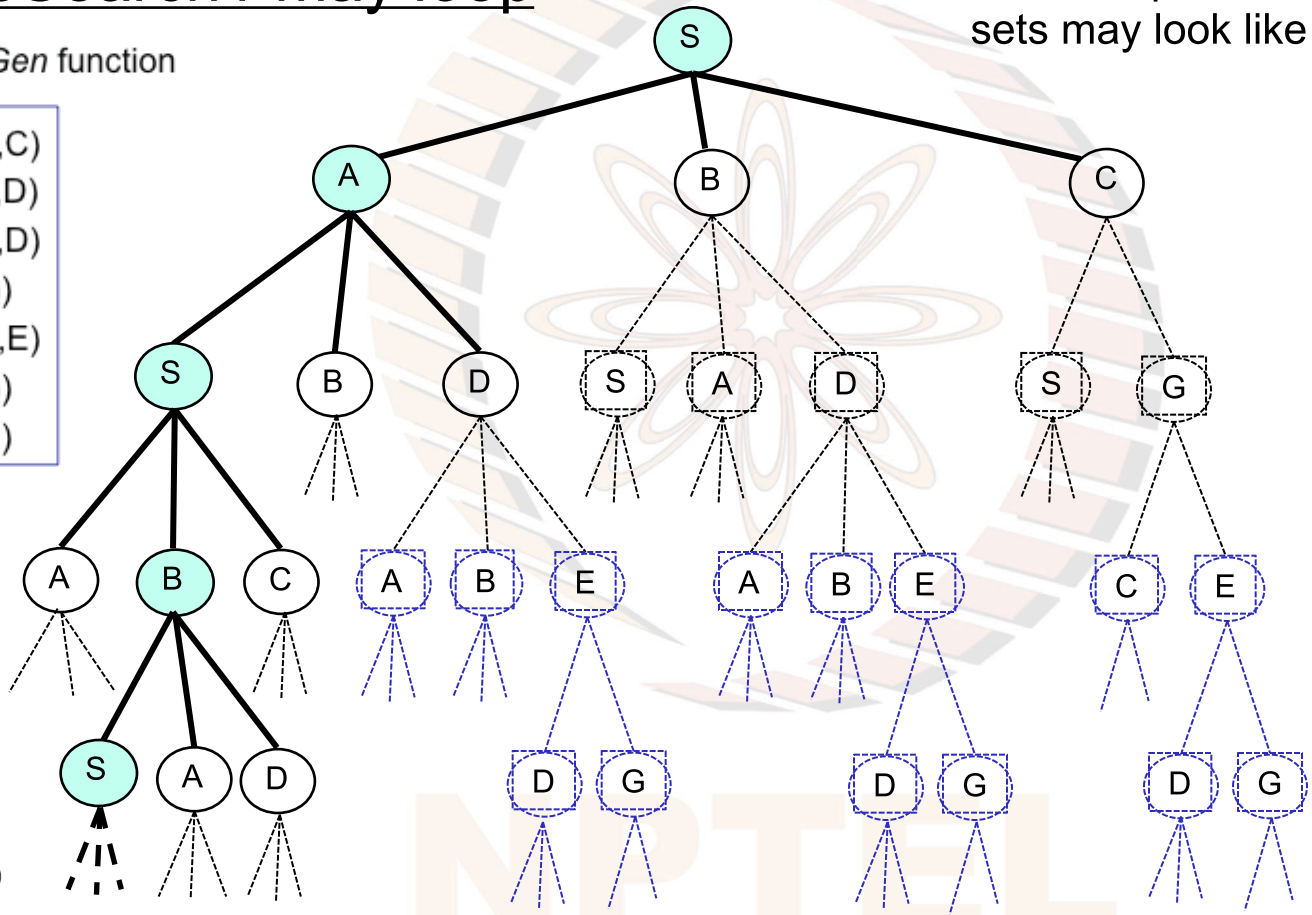
```
SimpleSearch1()
    1   OPEN ← {S}
    2   while OPEN is not empty
    3     do pick some node N from OPEN
    4              OPEN ← OPEN – {n}
    5              if GoalTest(N) = TRUE
    6                then   return N
    7                else   OPEN ← OPEN ∪ MoveGen(N)
    8   return FAILURE
```

Algorithm *SimpleSearch1*

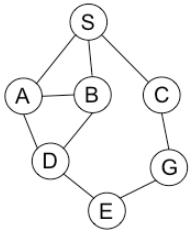
SimpleSearch1 may loop

The *MoveGen* function

S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



The State Space



For example the successive OPEN sets may look like

(**S**)
(**A**BC)
(**S**BDBC)
(A**B**CBDBC)
(A**S**ADCBDBC)

CLOSED: a repository of seen nodes

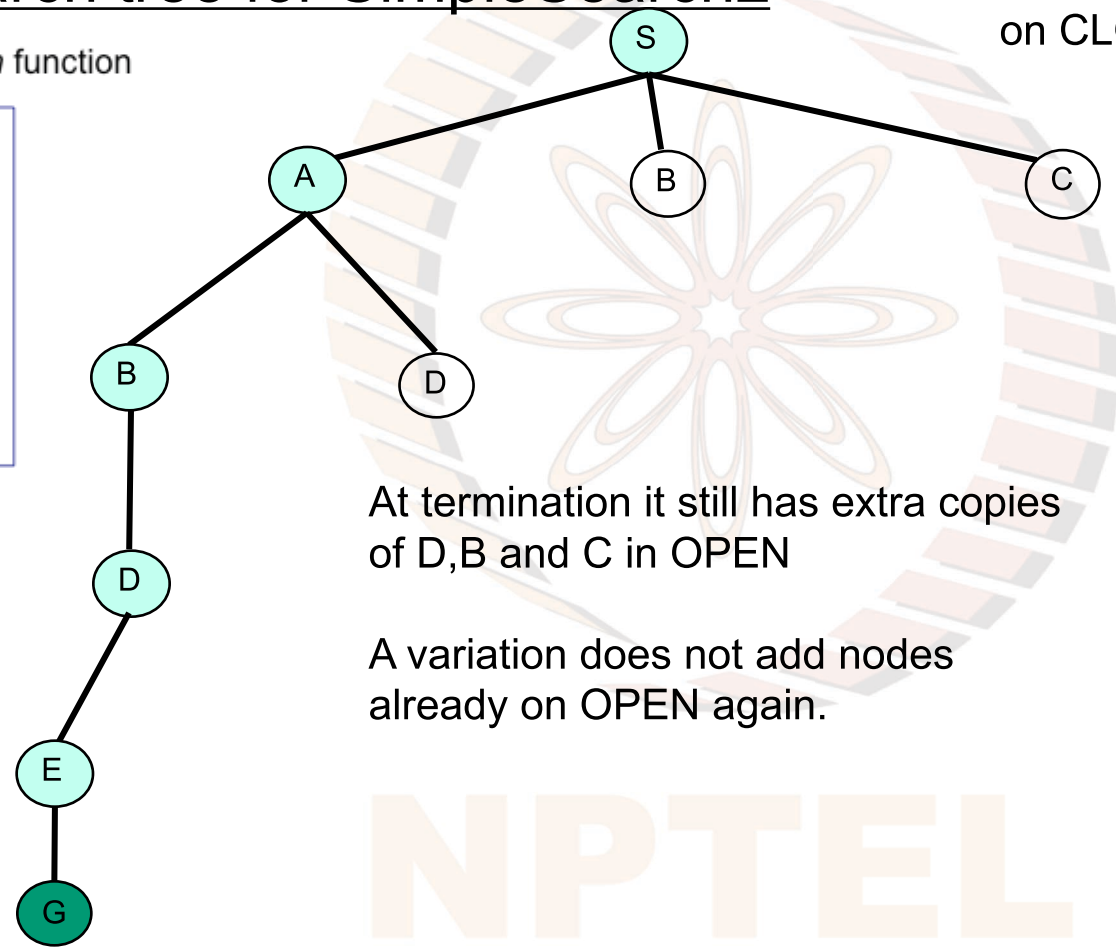
SimpleSearch2()
1 *OPEN* ← {*start*}
2 *CLOSED* ← { }
3 **while** *OPEN is not empty*
4 **do** Pick *some* node N from open
5 *OPEN* ← *OPEN* − {*N*}
6 *CLOSED* ← *CLOSED* ∪ {*N*}
7 **if** *GoalTest(N) = TRUE*
8 **then return** *N*
9 **else** *OPEN* ← *OPEN* ∪ {*MoveGen*(*N*) − *CLOSED*}
10 **return** *FAILURE*

Algorithm *SimpleSearch2*

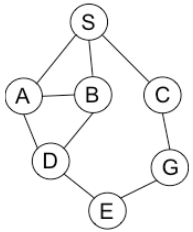
The search tree for SimpleSearch2

The *MoveGen* function

S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



The State Space



At termination it still has extra copies of D,B and C in OPEN

A variation does not add nodes already on OPEN again.

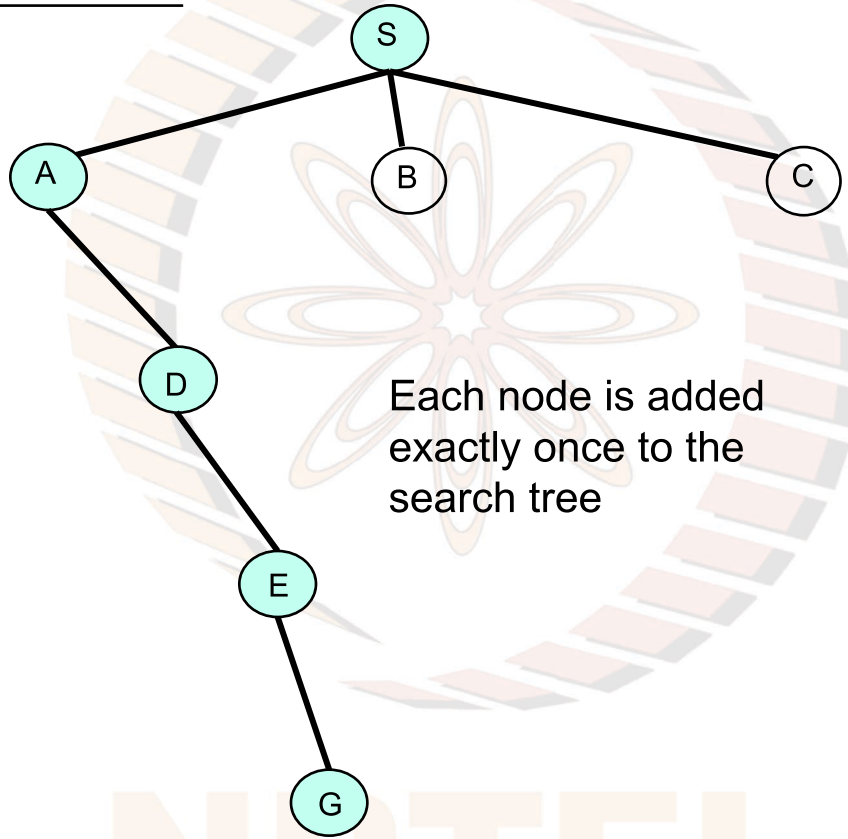
It does not add nodes on CLOSED to OPEN

OPEN	CLOSED
(**S**)	( )
(**A**BC)	(S)
(**B**DBC)	(AS)
(**D**DBC)	(BAS)
(**E**DBC)	(DBAS)
(**G**DBC)	(EDBAS)

Adding *only* new nodes

The *MoveGen* function

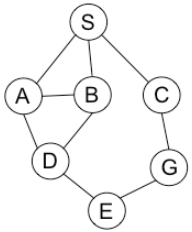
S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



Each node is added exactly once to the search tree

OPEN	CLOSED
(**S**)	( )
(**A**BC)	(S)
(**D**BC)	(AS)
(**E**BC)	(DAS)
(**G**BC)	(EDAS)

The State Space



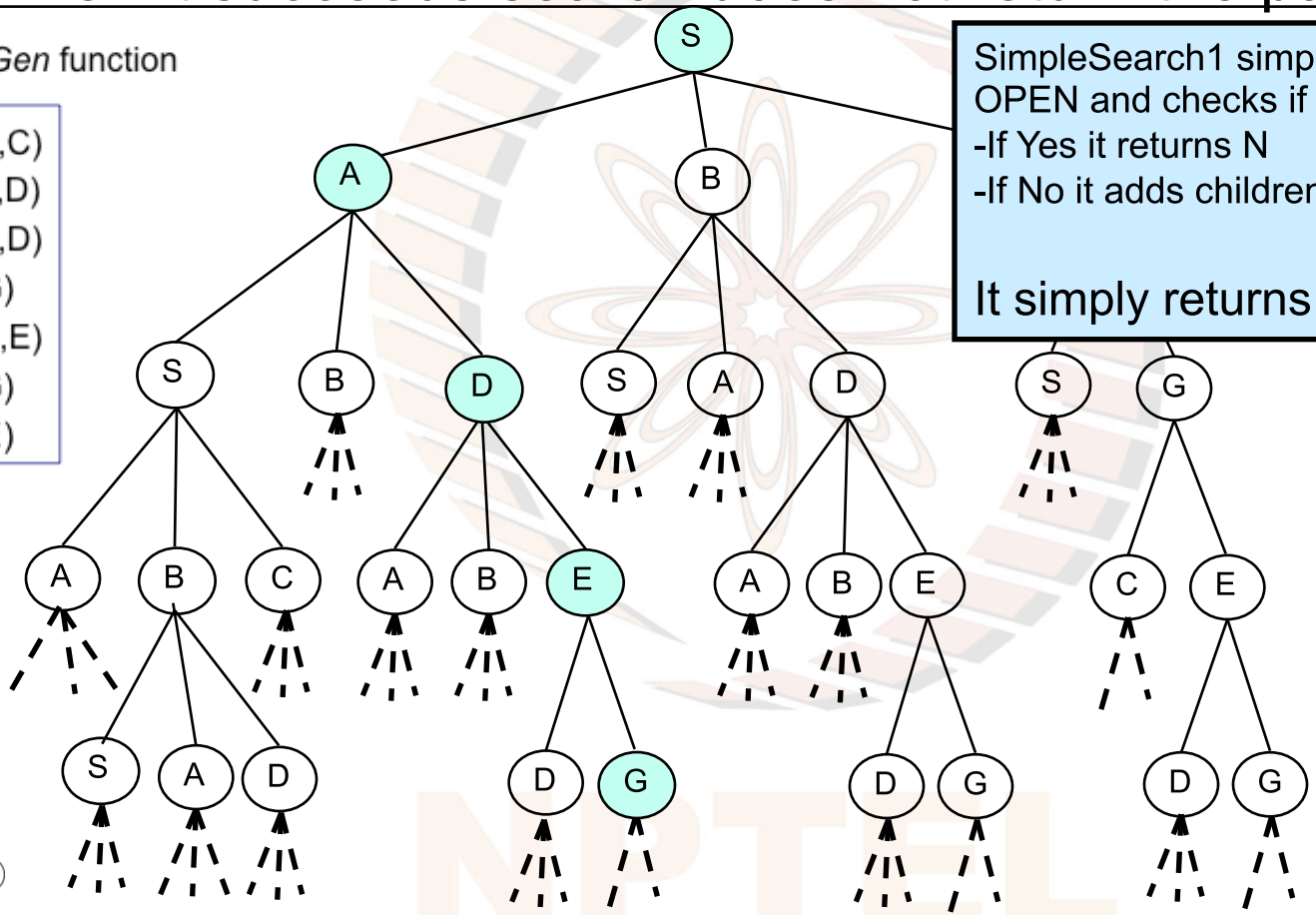
Even when it succeeds search does not return the path

The *MoveGen* function

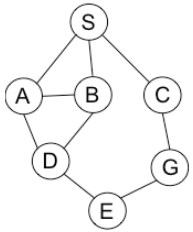
S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)

SimpleSearch1 simply picks a node N from OPEN and checks if it is the goal.
-If Yes it returns N
-If No it adds children of N to OPEN

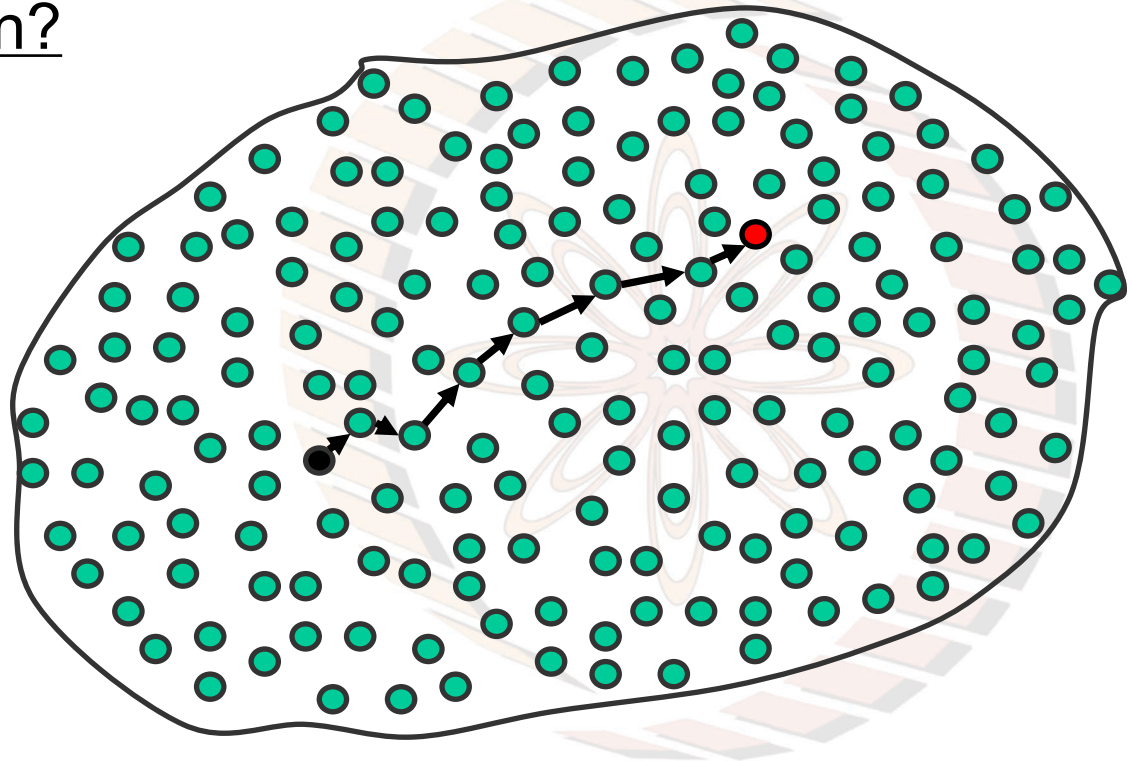
It simply returns node G!



The State Space



A Solution?



SimpleSearch2 returns the goal node when it finds it. What is needed in many problems is the PATH to the goal node.

Planning problems

Goal is known, path is needed

Some examples

River crossing problems

Route finding problems

Rubik's cube

8/15/24-puzzle

Cooking a dish

Configuration problems

A state satisfying a description is needed

Some examples

N-queens

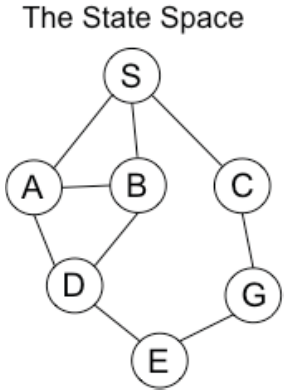
A crossword puzzle

Sudoku

Map colouring

SAT

NodePairs : Keep track of parent nodes



The State Space is modeled as a graph with each node being a state.

We need to keep track of the parents of each node during search so that we can reconstruct the path when we find the goal node.

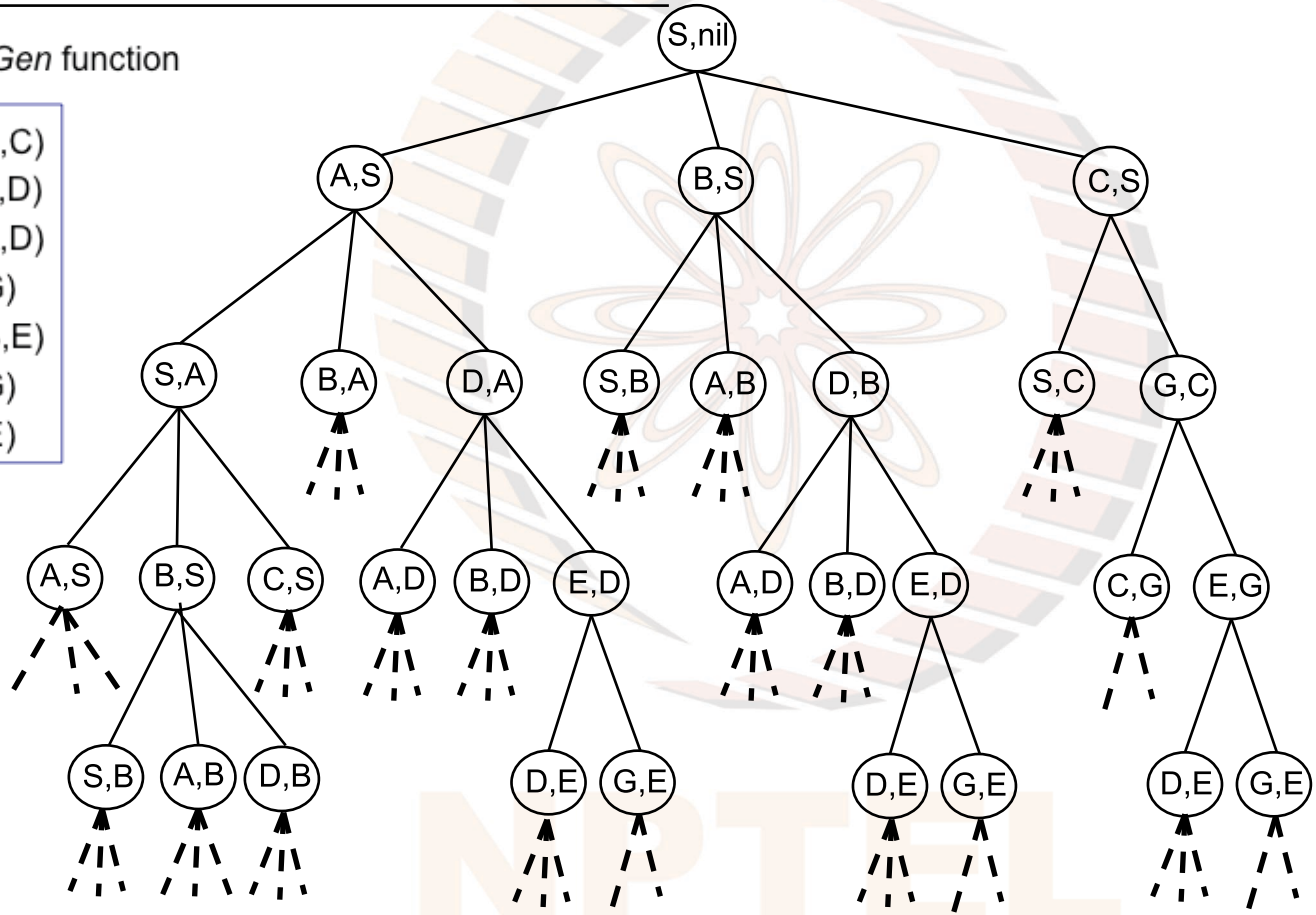
One way is to modify the search space node to store the entire path to the state space node itself.

A more elegant way is to store node pairs in the search space, where each node in search space is a pair (currentNode, parentNode)

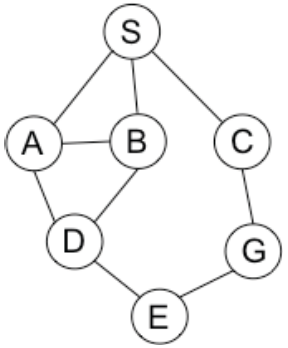
NodePairs in the Search Tree

The *MoveGen* function

S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



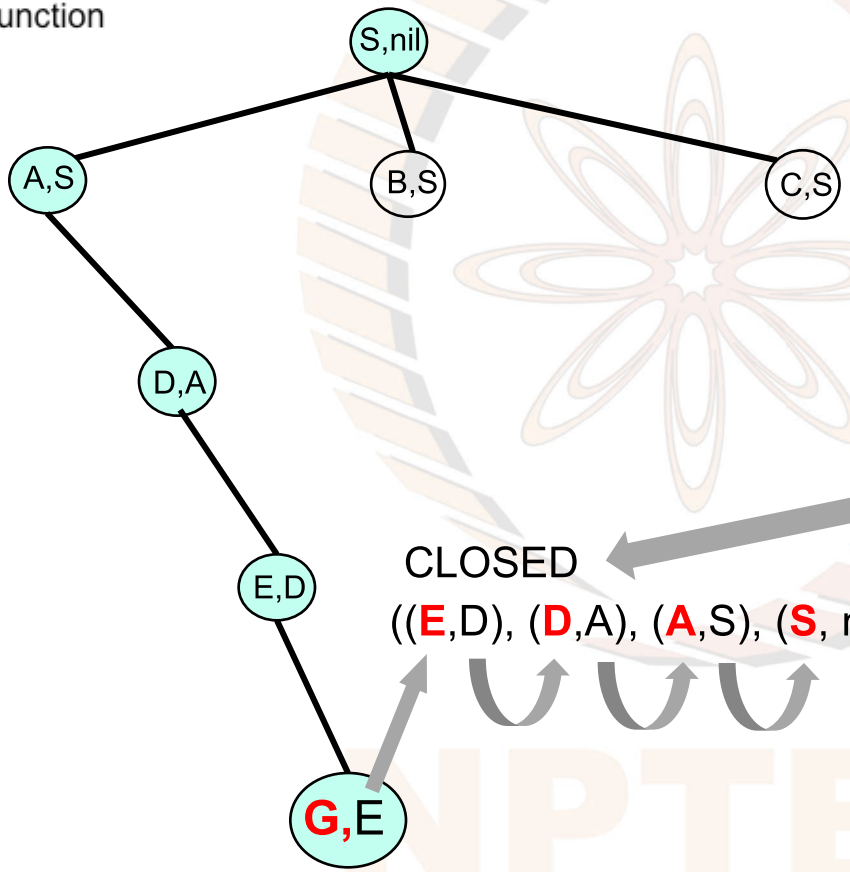
The State Space



Reconstructing the Path

The *MoveGen* function

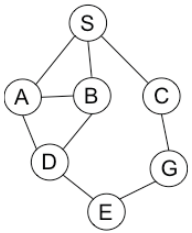
S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



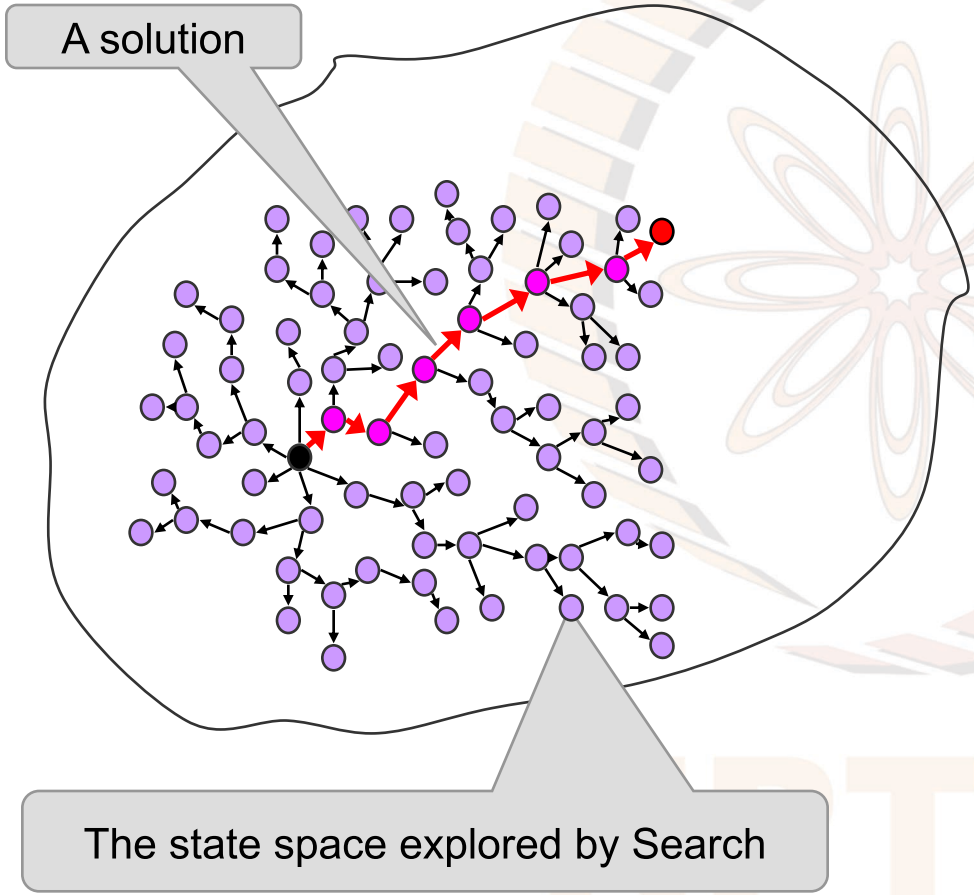
OPEN	CLOSED
((S,nil))	( )
((A,S),(B,S),(C,S))	((S,nil))
((D,A),(B,S),(C,S))	((A,S),(S,nil))
((E,D),(B,S),(C,S))	((D,A),(A,S), (S,nil))
((G,E),(B,S),(C,S))	((E,D),(D,A), (A,S), (S,nil))

CLOSED
((E,D), (D,A), (A,S), (S, nil))

The State Space



A Search Program Generates and Explores a Search Tree



pick *some* node *N* from OPEN

.

.

.

OPEN ← *OPEN* ∪ *MoveGen(N)*

Non-determinism

Deterministic Search Algorithms

- Use LIST data structures instead of SET
- Replace
  "pick *some* node *N* from OPEN"
  with
  "pick node from head of OPEN"
- Instead of adding new nodes to OPEN as a SET insert them in a specified place in the list.
  - where you do so will determine the behaviour of the search algorithm

pick *some* node *N* from OPEN
.
.
.
OPEN ← *OPEN* ∪ *MoveGen(N)*

Non-determinism

Some LIST notation

1. [ ] is an empty list

2. [ ] **is empty** $=$ TRUE

3. [1] **is empty** $=$ FALSE

4. $\text{LIST}_2 \leftarrow \text{ELEMENT}$ **:** LIST_1

5. $\text{LIST}_2 \leftarrow \text{HEAD}$ **:** TAIL

Some LIST notation (continued)

$$\text{LIST}_2 \leftarrow \text{ELEMENT} : \text{LIST}_1$$
$$\text{LIST}_2 \leftarrow \text{HEAD} : \text{TAIL}$$

6. $[1] = 1 : [\,]$

7. $1 = \textbf{head}\,[1] = \textbf{head}\,1 : [\,]$

8. $[\,] = \textbf{tail}\,[1] = \textbf{tail}\,1 : [\,]$

9. $(\textbf{tail}\,[1])\ \textbf{is empty} = \text{TRUE}$

Some LIST notation (continued)

10. $[3,2,1] = 3 : [2,1] = 3 : 2 : [1] = 3 : 2 : 1 : [\,]$

11. $3 = \textbf{head}\ [3,2,1]$

12. $[2,1] = \textbf{tail}\ [3,2,1]$

13. $2 = \textbf{head tail}\ [3,2,1]$

14. $1 = \textbf{head tail tail}\ [3,2,1]$

Some LIST notation (continued)

15. $\text{LIST}_3 = \text{LIST}_1 \mathbin{+\!\!+} \text{LIST}_2$

16. $[\,] = [\,] \mathbin{+\!\!+} [\,]$

17. $\text{LIST} = \text{LIST} \mathbin{+\!\!+} [\,] = [\,] \mathbin{+\!\!+} \text{LIST}$

18. $[o, u, t, r, u, n] = [o, u, t] \mathbin{+\!\!+} [r, u, n]$

19. $[r, u, n, o, u, t] = [r, u, n] \mathbin{+\!\!+} [o, u, t]$

20. $[r, o, u, t] = (\textbf{head}\,[r, u, n])\,\textbf{:}\,[o, u, t]$

21. $[n, u, t] = \textbf{tail tail}\,[r, u, n] \mathbin{+\!\!+} \textbf{tail}\,[o, u, t]$

Some TUPLE operators

1. $(a, b) \leftarrow (101, 102)$

2. pair $\leftarrow (101, 102)$

3. $(a, b) \leftarrow$ pair

4. $(a, \_) \leftarrow$ pair

5. a $\leftarrow$ **first** pair

Some TUPLE operators

6. $(\_, b) \leftarrow$ pair

7. $b \leftarrow$ **second** pair

8. $(\_, \_, c) \leftarrow (101, \text{'AI SMPS'}, 4)$

9. $c \leftarrow$ **third** $(101, \text{'AI SMPS'}, 4)$

10. $4 =$ **third** $(101, \text{'AI SMPS'}, 4)$

Lists and Tuples

11. $101 \quad = \quad \textbf{head second}\ (1, [101, 102, 103], \textbf{null})$

12. $[102, 103] \quad = \quad \textbf{tail second}\ (1, [101, 102, 103], \textbf{null})$

13. $(a, h : t, c) \quad \leftarrow \quad (1, [101, 102, 103], \textbf{null})$

$a = 1$

$h = 101$

$t = [102, 103]$

$c = \textbf{null}$

Depth First Search: OPEN = Stack data structure

```
DFS(S)
 1  OPEN ← (S, null) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4        nodePair ← head OPEN
 5        (N, _) ← nodePair
 6        if GoalTest(N) = true
 7              return ReconstructPath(nodePair, CLOSED)
 8        else CLOSED ← nodePair : CLOSED
 9              children ← MoveGen(N)
10              newNodes ← RemoveSeen(children, OPEN, CLOSED)
11              newPairs ← MakePairs(newNodes, N)
12              OPEN ← newPairs ++ (tail OPEN)
13  return empty list
```

Ancillary functions : remove duplicates

```
RemoveSeen(nodeList, OPEN, CLOSED)
1  if nodeList is empty
2        return empty list
3  else node ← head nodeList
4        if OccursIn(node, OPEN) or OccursIn(node, CLOSED)
5              return RemoveSeen(tail nodeList, OPEN, CLOSED)
6        else return node : RemoveSeen(tail nodeList, OPEN, CLOSED)
```

Removes from nodeList any node that are already in OPEN or in CLOSED

Ancillary functions : occursIn

```
OccursIn(node, nodePairs)
1  if nodePairs is empty
2        return FALSE
3  elseif node = first head nodePairs
4        return TRUE
5  else return OccursIn(node, tail nodePairs)
```

Looks for node N as the first element of a nodePair in a list of nodePairs. Used for removing duplicates

Ancillary functions : makePairs

MakePairs(nodeList, parent)

Converts a list of nodes to a list of node pairs, with *parent* being the parent of each node in list

```
1  if nodeList is empty
2        return empty list
3  else (head nodeList, parent) : MakePairs(tail nodeList, parent)
```

Makes a pair (head(nodeList), parent)

For example
makePairs((A, B, C, D), S) = ((A,S), (B,S), (C,S), (D,S))

Ancillary functions

ReconstructPath(nodePair, CLOSED)

Traces back through Parent links and reconstructs the path found.

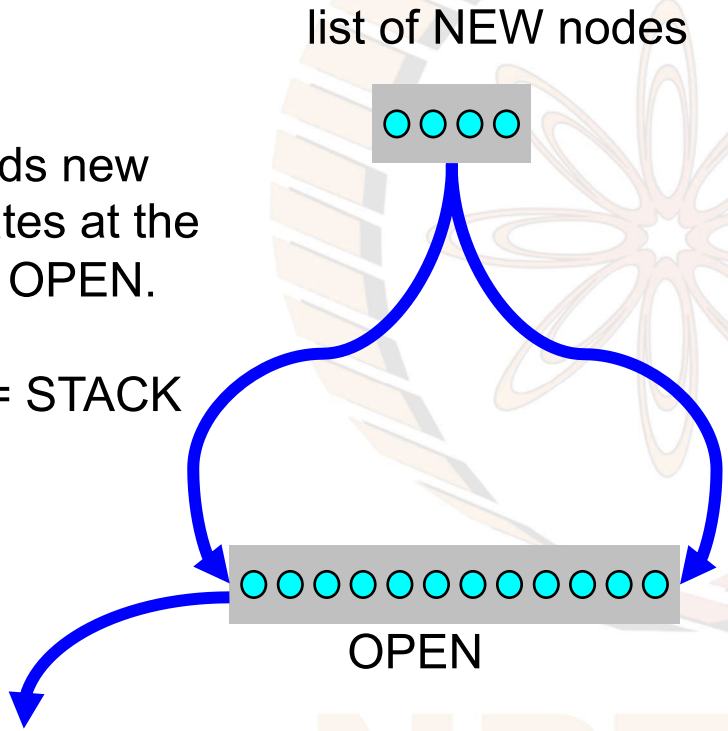
```
1  SkipTo(parent, nodePairs)
2        if parent = first head nodePairs
3              return nodePairs
4        else return SkipTo(parent, tail nodePairs)

5  (node, parent) ← nodePair
6  path ← node : [ ]
7  while parent is not null
8        path ← parent : path
9        CLOSED ← SkipTo(parent, CLOSED)
10       (_, parent) ← head CLOSED
11 return path
```

Breadth First Search: OPEN = Queue

```
BFS(S)
 1  OPEN ← (S, null) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4        nodePair ← head OPEN
 5        (N, _) ← nodePair
 6        if GoalTest(N) = true
 7              return ReconstructPath(nodePair, CLOSED)
 8        else CLOSED ← nodePair : CLOSED
 9              children ← MoveGen(N)
10              newNodes ← RemoveSeen(children, OPEN, CLOSED)
11              newPairs ← MakePairs(newNodes, N)
12              OPEN ← (tail OPEN) ++ newPairs
13  return empty list
```

Stack vs. Queue



DFS adds new candidates at the head of OPEN.

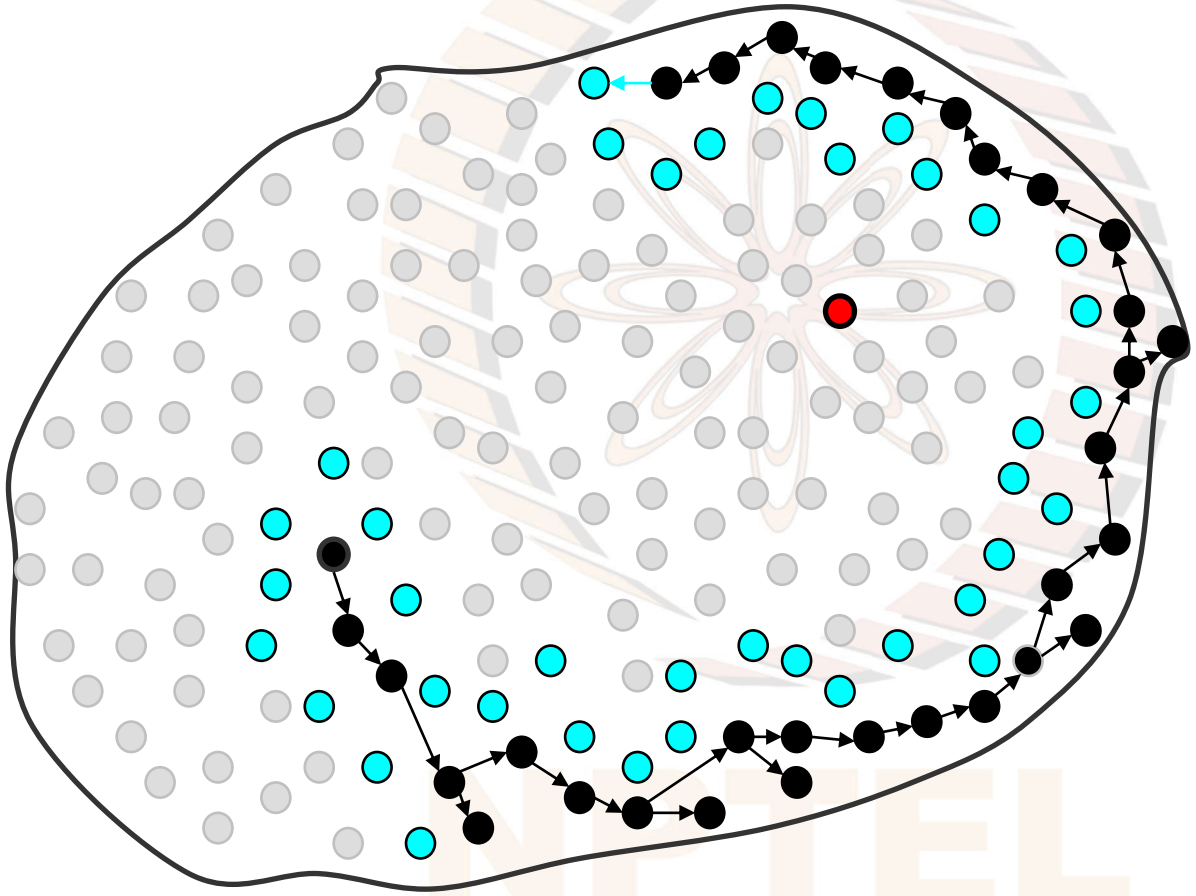
OPEN = STACK

BFS adds new candidates at the head of OPEN.

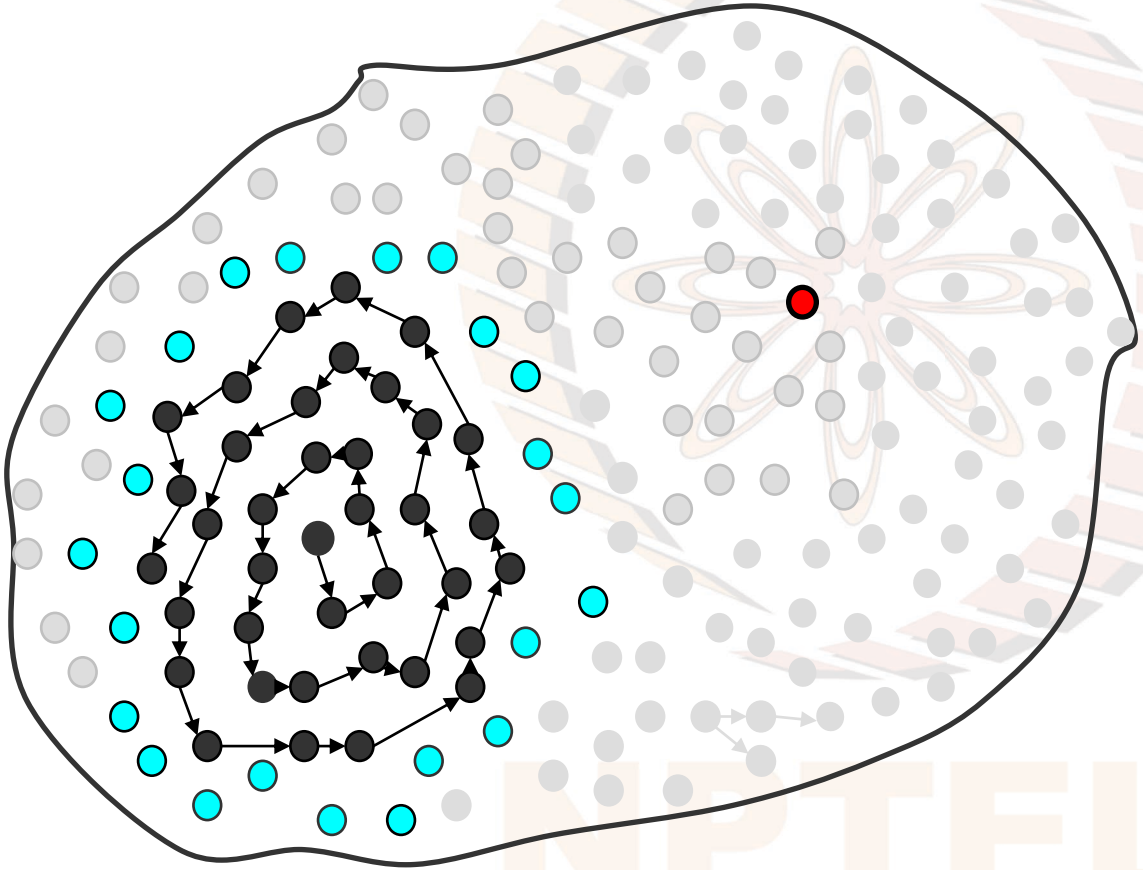
OPEN = QUEUE

- Search picks candidate from head of OPEN

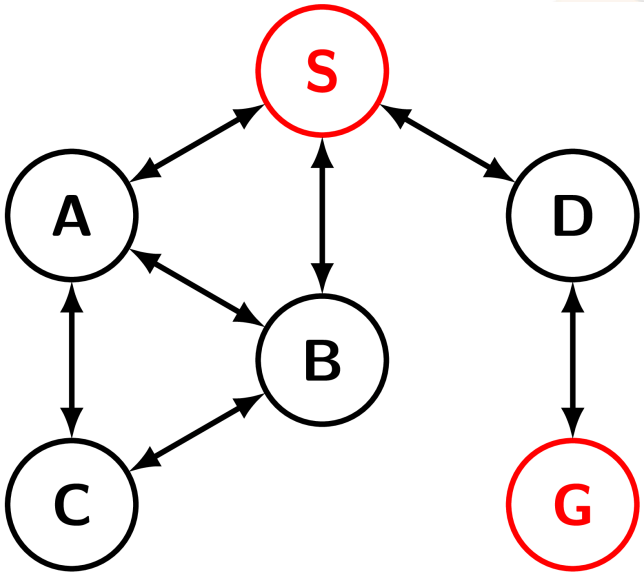
Depth First Search



Breadth First Search

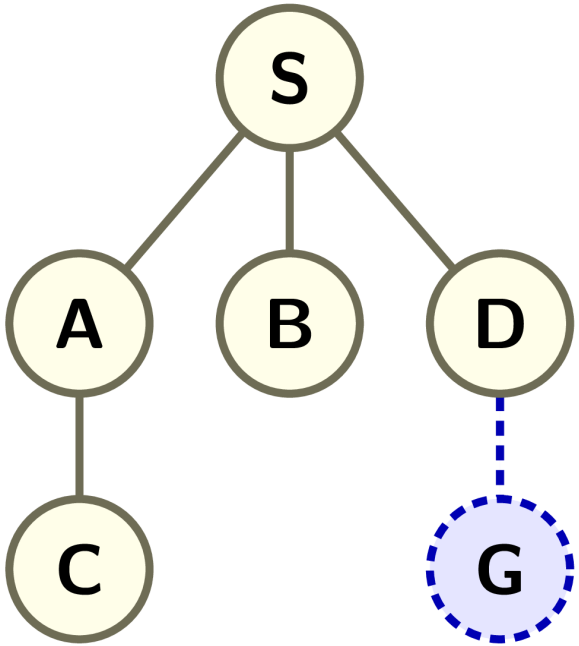


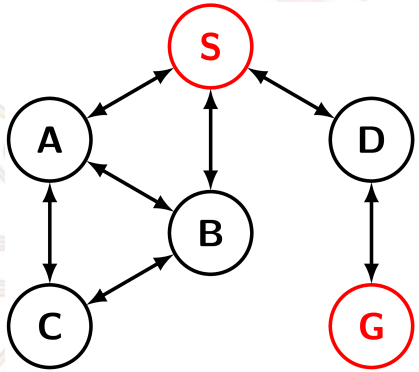
A Tiny State Space



X		MoveGen(X)	GoalTest(X)
S	→	A,B,D	False
A	→	C,B,S	False
B	→	S,A,C	False
C	→	B,A	False
D	→	S,G	False
G	→	D	True

Case 1 : Only new nodes are added to OPEN

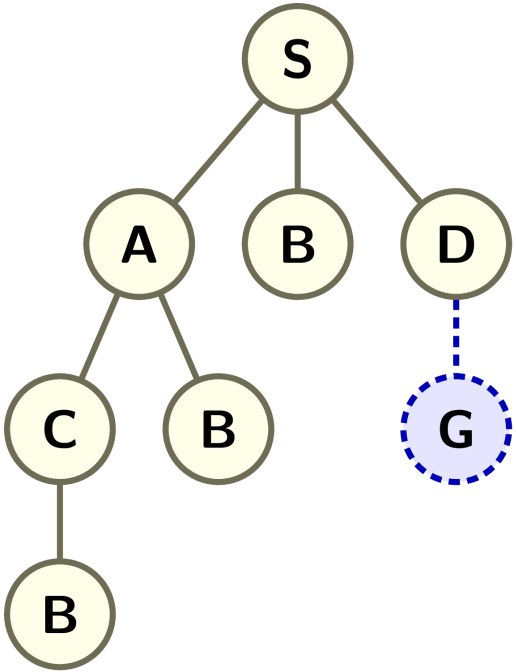




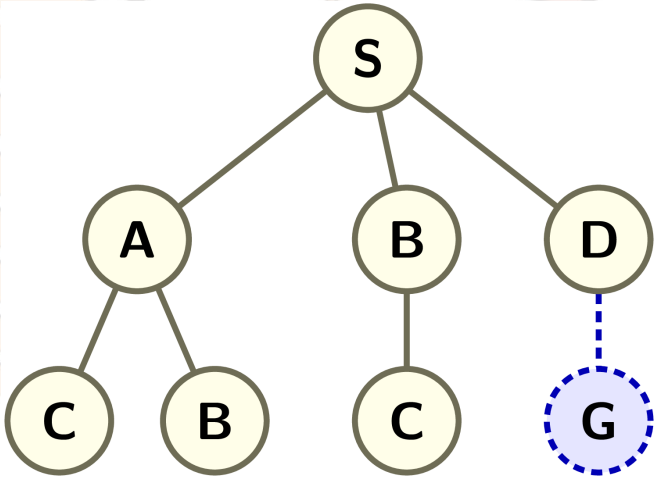
X		MoveGen(X)	GoalTest(X)
S	→	A,B,D	False
A	→	C,B,S	False
B	→	S,A,C	False
C	→	B,A	False
D	→	S,G	False
G	→	D	True

Both DFS and BFS explore the same search tree.
The order of exploration is different though.

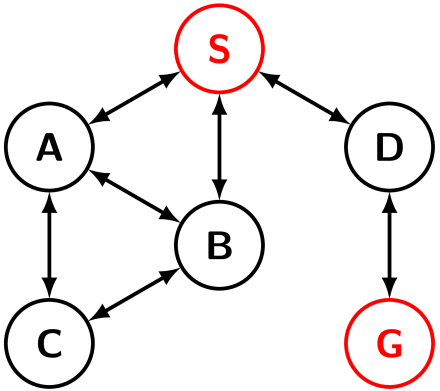
Case 2 : Nodes already in CLOSED are pruned



DFS search tree

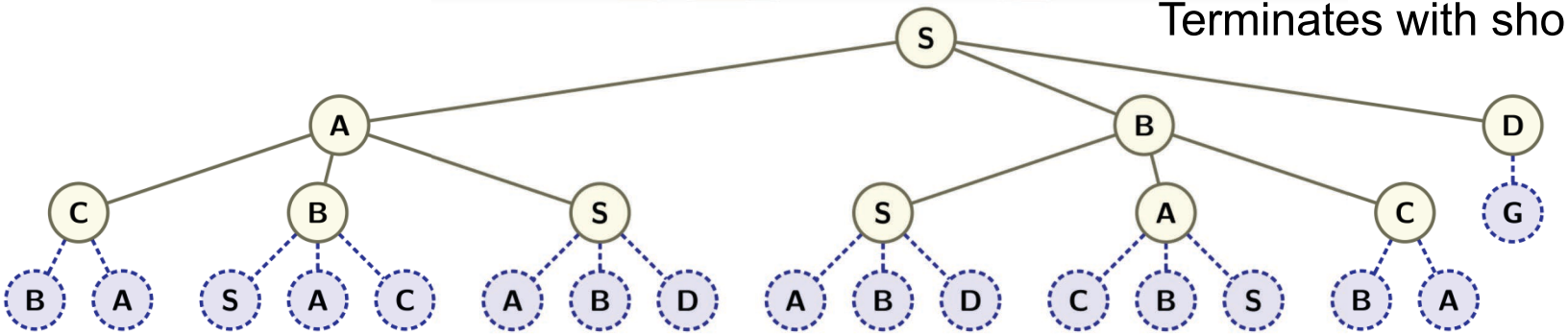


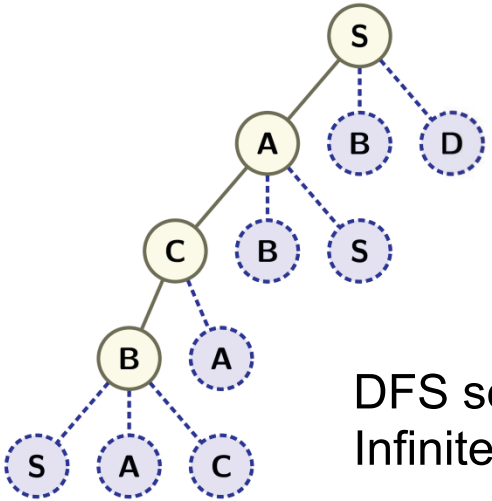
BFS search tree

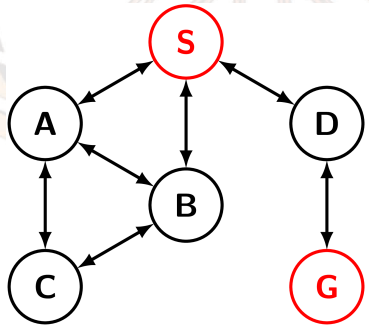


Case 3 : All nodes are added again

BFS search tree.
Terminates with shortest path

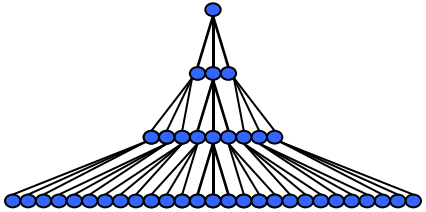






DFS search tree
Infinite loop

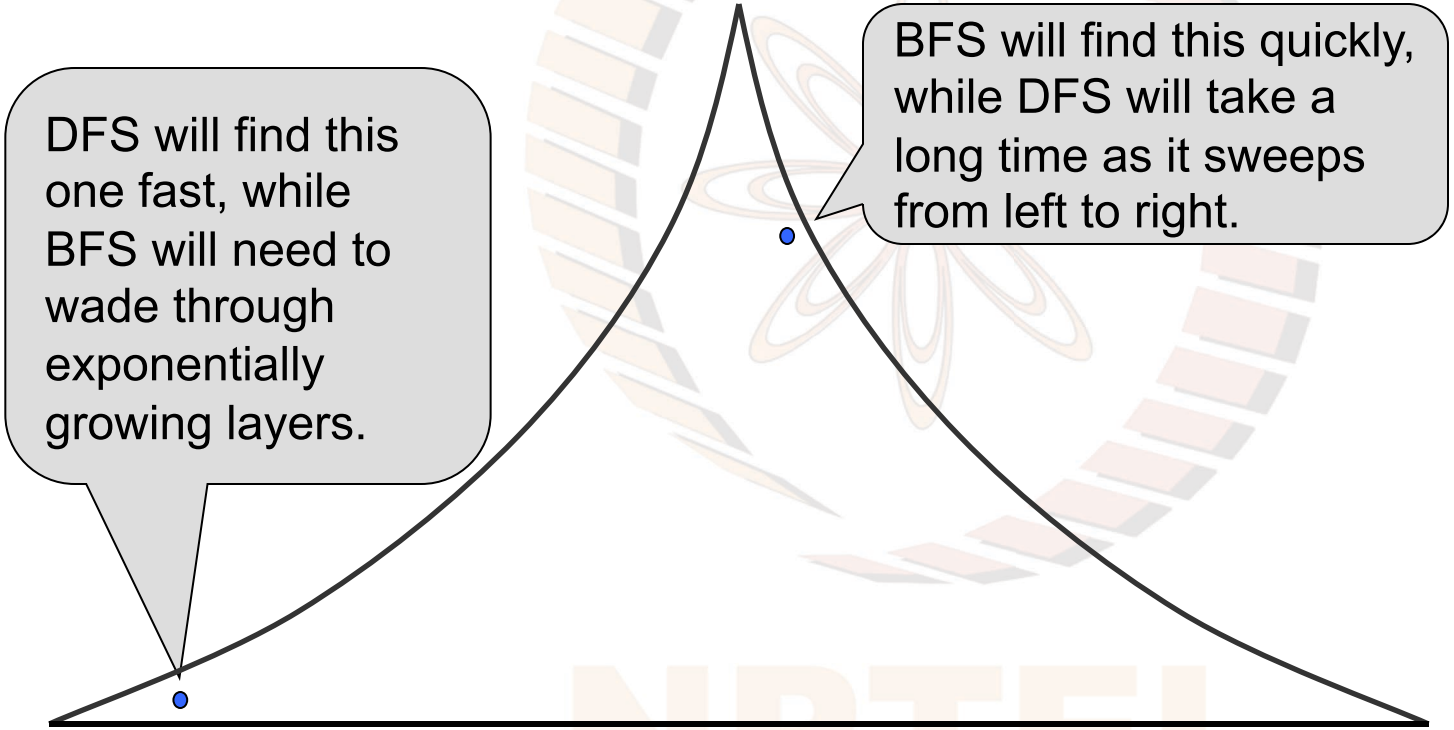
DFS and BFS : Behavior



A Search Tree

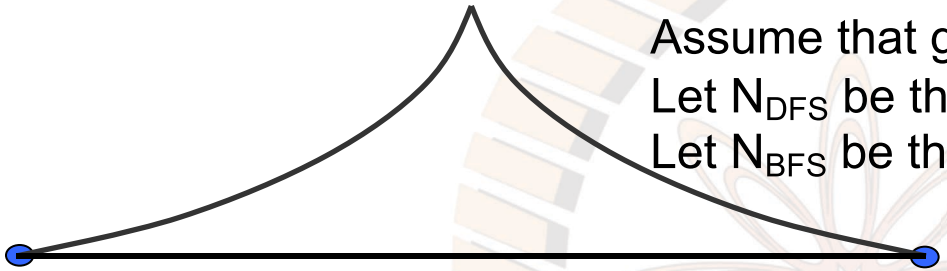
When the two searches pick the fifth node, coloured green, they have explored different parts of the search tree, expanding the four nodes before in the order shown.

DFS vs BFS : Running Time



Time complexity

Assume constant branching fact b
Assume that goal occurs somewhere at depth d
Let N_{DFS} be the number of nodes inspected by DFS
Let N_{BFS} be the number of nodes inspected by BFS



$N_{DFS} = d+1$
$N_{BFS} = (b^d-1)/(b-1) +1$

$N_{DFS} = N_{BFS} = (b^{d+1}-1)/(b-1)$

On the average

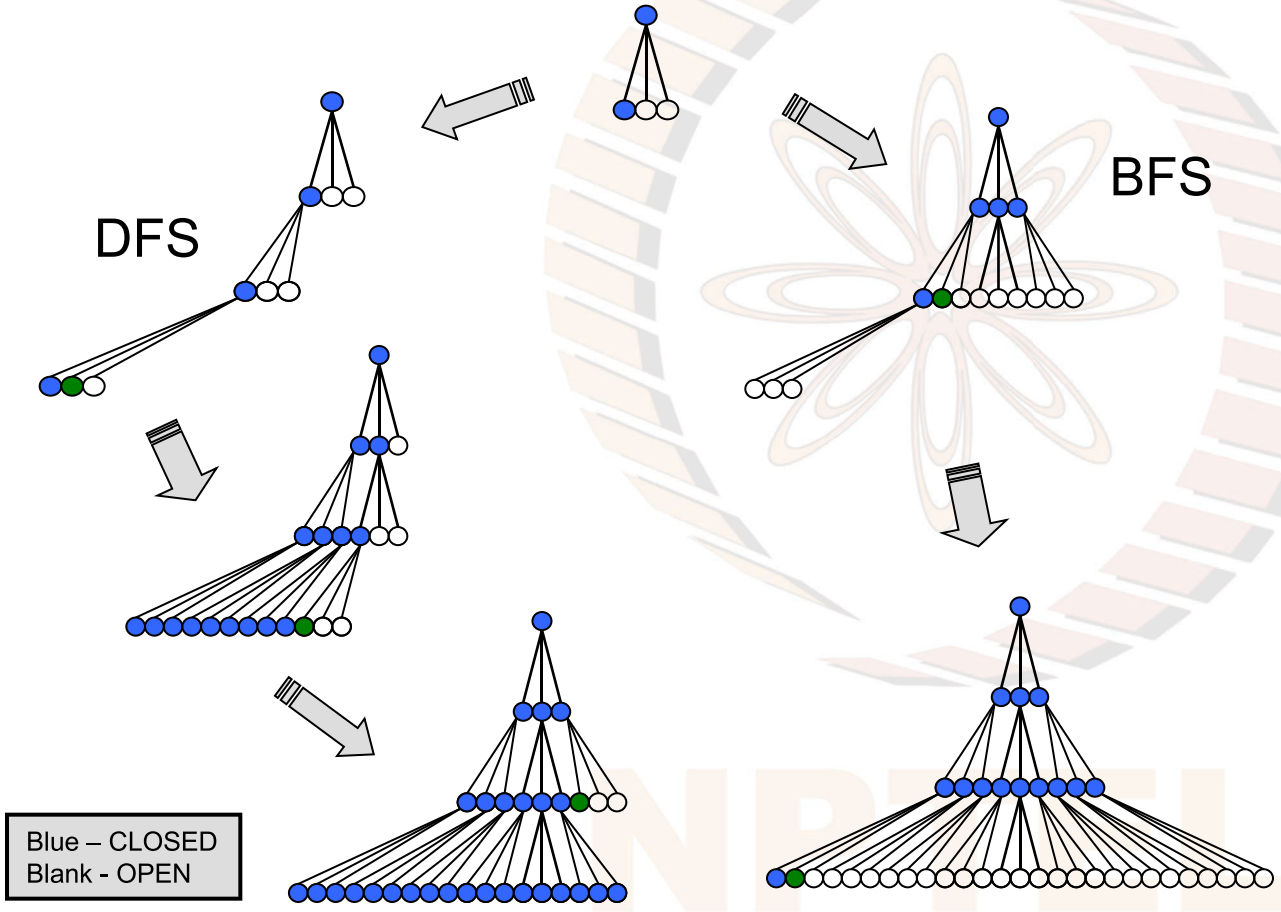
$$N_{DFS} = \frac{(d+1) + (b^{d+1}-1)/(b-1)}{2} \approx \frac{b^d}{2}$$

$$N_{BFS} = \frac{(b^d -1)/(b-1) +1 + (b^{d+1}-1)/(b -1)}{2} \approx \frac{b^d(b+1)}{2(b-1)}$$

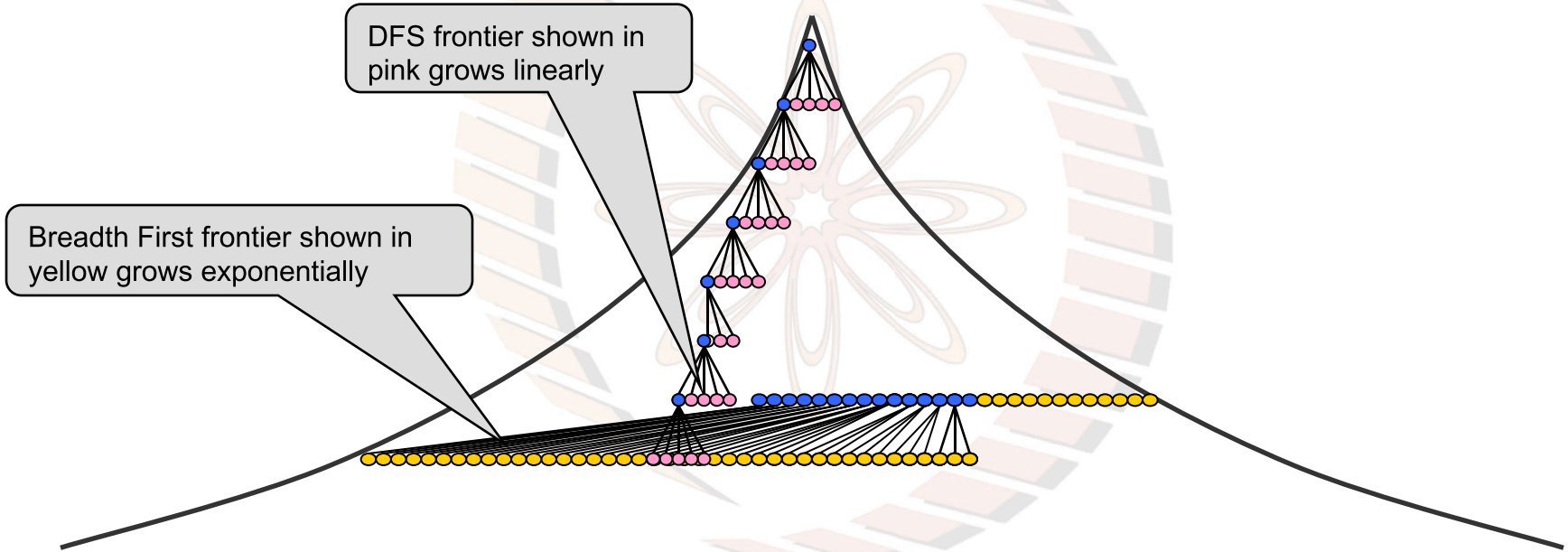
$$N_{BFS} \approx N_{DFS} \frac{(b+1)}{(b-1)}$$

Average time complexity is exponential

DFS vs BFS : Space (Size of OPEN)

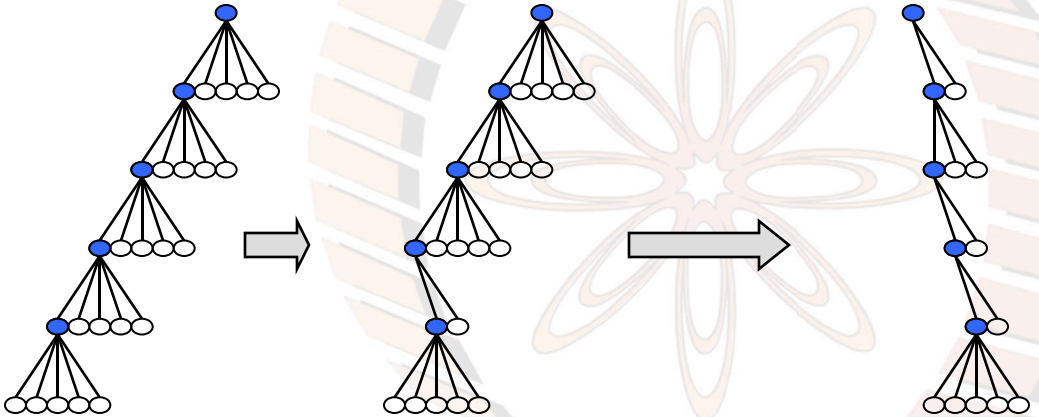


Search Frontiers



The number of nodes on the search frontier is an indication of space requirement

DFS : The OPEN list



With a branching factor of 5, at depth 5 there are 5(5-1) +1 = 21 nodes in the OPEN to begin with.

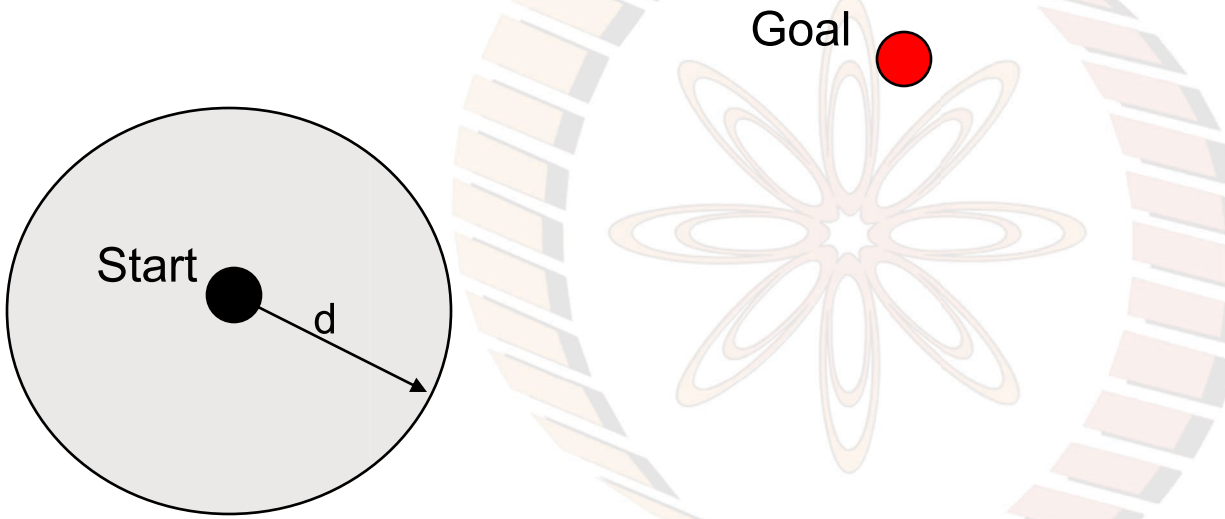
But as DFS progresses and the algorithm backtracks and moves right, the number of nodes on OPEN decrease.

Depth First vs. Breadth First

	Depth First Search	Breadth First Search
Time	Exponential	Exponential
Space	Linear	Exponential
Quality of solution	No guarantees	Shortest path
Completeness	Not for infinite search space	Guaranteed to terminate if solution path exists

Can we devise a algorithm that uses linear space for OPEN and guarantees and shortest path?

Depth Bounded DFS (DBDFS)



Do DFS with a depth bound *d*.

- Linear space
- Not complete
- Does not guarantee shortest path

Depth Bounded DFS : Store depth information in search node

```
DB-DFS(S, depthBound)
 1  OPEN ← (S, null, 0) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4        nodePair ← head OPEN
 5        (N, _, depth) ← nodePair
 6        if GoalTest(N) = true
 7              return ReconstructPath(nodePair, CLOSED)
 8        else CLOSED ← nodePair : CLOSED
 9              if depth < depthBound
10                    children ← MoveGen(N)
11                    newNodes ← RemoveSeen(children, OPEN, CLOSED)
12                    newPairs ← MakePairs(newNodes, N, depth + 1)
13                    OPEN ← newPairs ++ tail OPEN
14              else OPEN ← tail OPEN
15  return empty list
```

DBDFS-2: returns count of nodes visited

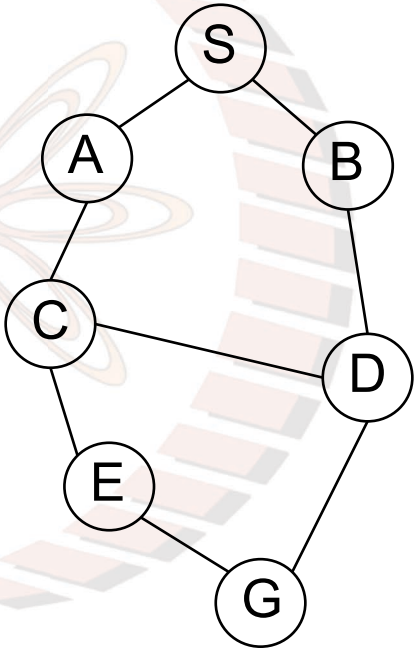
```
DB-DFS-2(S, depthBound)
 1  count ← 0
 2  OPEN ← (S, null, 0) : [ ]
 3  CLOSED ← empty list
 4  while OPEN is not empty
 5      nodePair ← head OPEN
 6      (N, _, depth) ← nodePair
 7      if GoalTest(N) = true
 8          return (count, ReconstructPath(nodePair, CLOSED))
 9      else CLOSED ← nodePair : CLOSED
10          if depth < depthBound
11              children ← MoveGen(N)
12              newNodes ← RemoveSeen(children, OPEN, CLOSED)
13              newPairs ← MakePairs(newNodes, N, depth + 1)
14              OPEN ← newPairs ++ tail OPEN
15              count ← count + length newPairs
16          else OPEN ← tail OPEN
17  return (count, empty list)
```

Depth First Iterative Deepening (DFID)

```
DFID(S)
1  count ← −1
2  path ← empty list
3  depthBound ← 0
4  repeat
5       previousCount ← count
6       (count, path) ← DB-DFS-2(S, depthBound)
7       depthBound ← depthBound + 1
8  until (path is not empty) or (previousCount = count)
9  return path
```

Another Tiny Search Problem

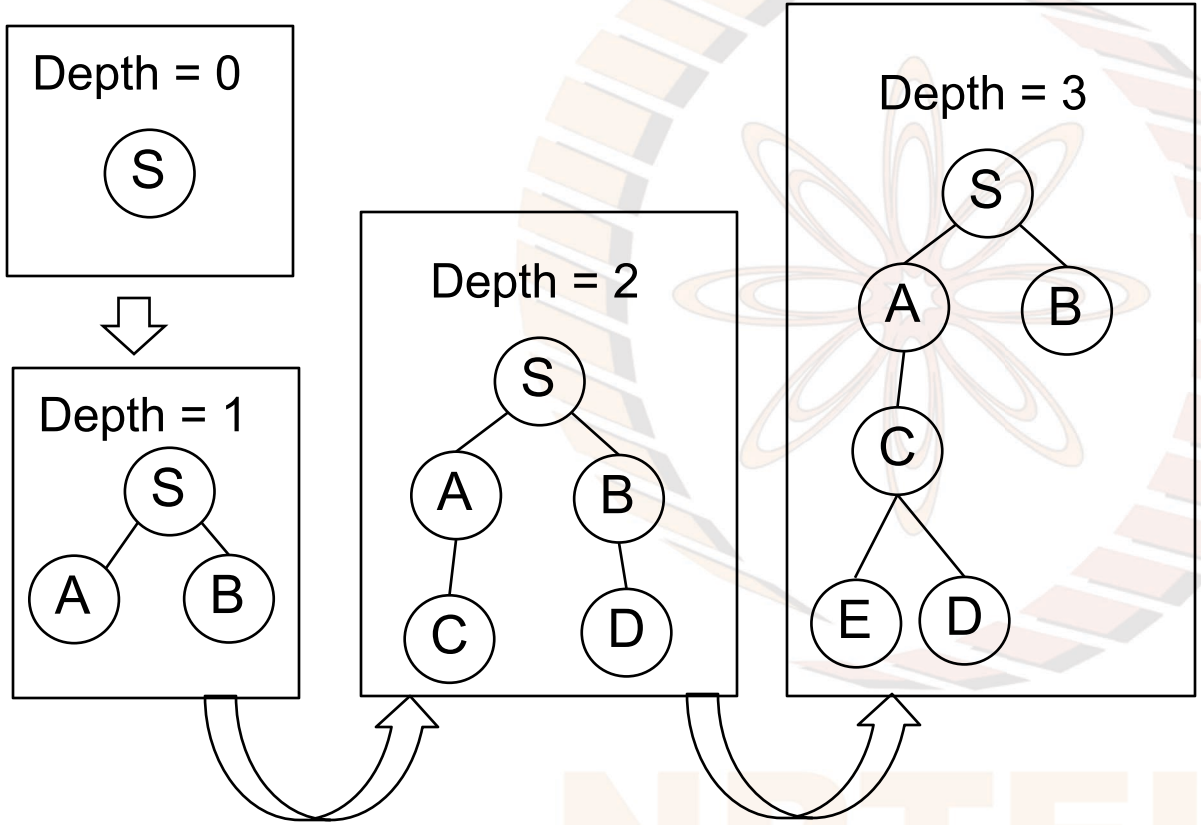
The State Space



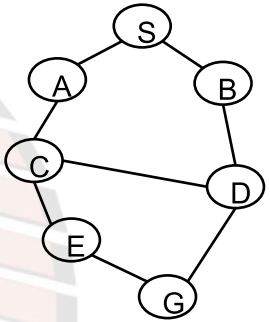
The *MoveGen* function

S → (A,B)
A → (S,C)
B → (S,D)
C → (E,D)
D → (B,C,G)
E → (C,G)
G → (D,E)

Careful with CLOSED in DFID

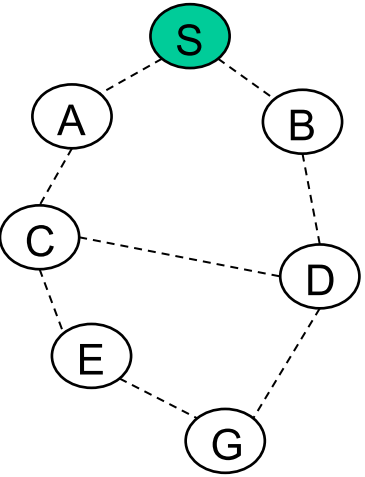


The State Space

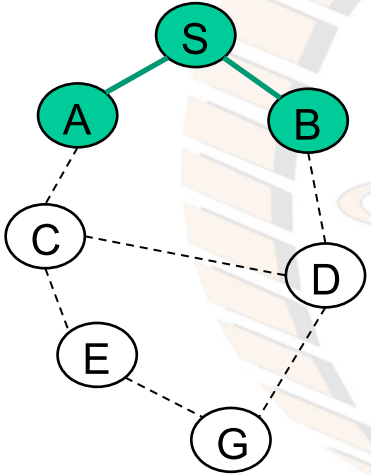


When depth = 3 D is not generated as a child of B!

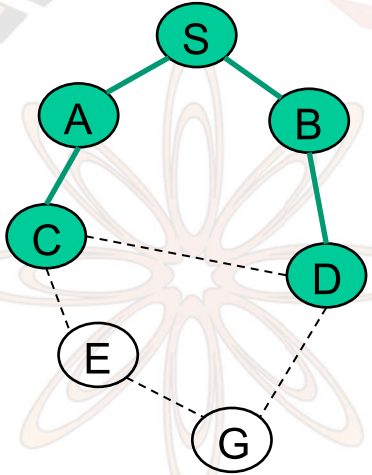
DFID checking loops with CLOSED



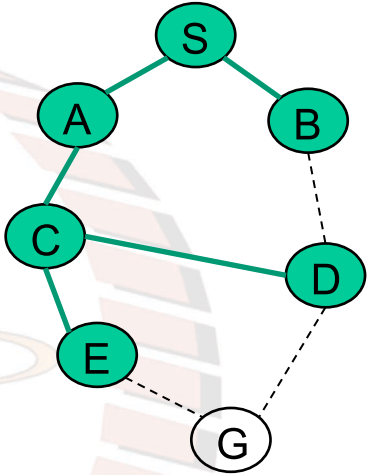
Path length 0



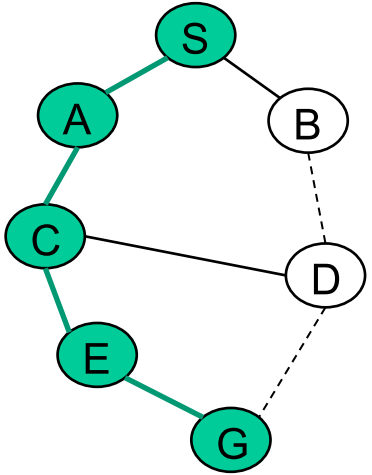
Path length 1



Path length 2



Path length 3



Path length 4

DFID first finds a path of length 3 to node D via S-A-C and blocks the path of length 2 via S-B and hence does not find the shortest path A-B-D-G to the goal

DB-DFS without CLOSED

```
DB-DFS(S, depthBound)
 1  count ← 0
 2  OPEN ← (S, null, 0, OPENED) : [ ]
 3  while OPEN is not empty
 4      (node, parent, depth, nodeStatus) ← head OPEN
 5      if GoalTest(node) = TRUE
 6          return (count, ReconstructPath(node, OPEN))
 7      elseif nodeStatus = CLOSED
 8          OPEN ← tail OPEN
 9      elseif depth < depthBound
10          OPEN ← tail OPEN
11          OPEN ← (node, parent, depth, CLOSED) : OPEN
12          children ← MoveGen(node)
13          newNodes ← RemoveSeen(children, OPEN)
14          newPairs ← MakePairs(newNodes, node, depth, OPENED)
15          OPEN ← newPairs ++ OPEN
16          count ← count + length newPairs
17      else OPEN ← tail OPEN
18  return (count, empty list)
```

In DB-DFS, there is no closed list, OPENED/CLOSED status is maintained in the open-list. Open-list acts like a call-stack. A node is maintained in the open-list (stack) for as long as its children are in the open-list (stack). A node is removed only after all its children are removed.

An OPENED node is removed from open-list (line 10) and immediately added back with CLOSED status (line 11) and only then its children are OPENED (lines 14, 15). A node is removed when its status is CLOSED (line 8) or when depth is out of bound (line 17).

Depth First Iterative Deepening (DFID)

```
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

Is there a catch?

A series of depth bounded depth first searches

(thus requiring linear space)

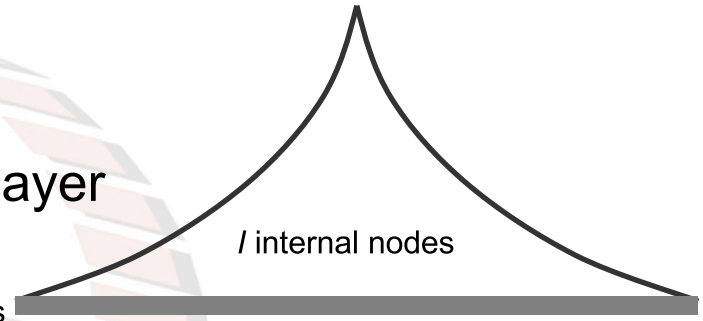
of increasing depth bound.

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

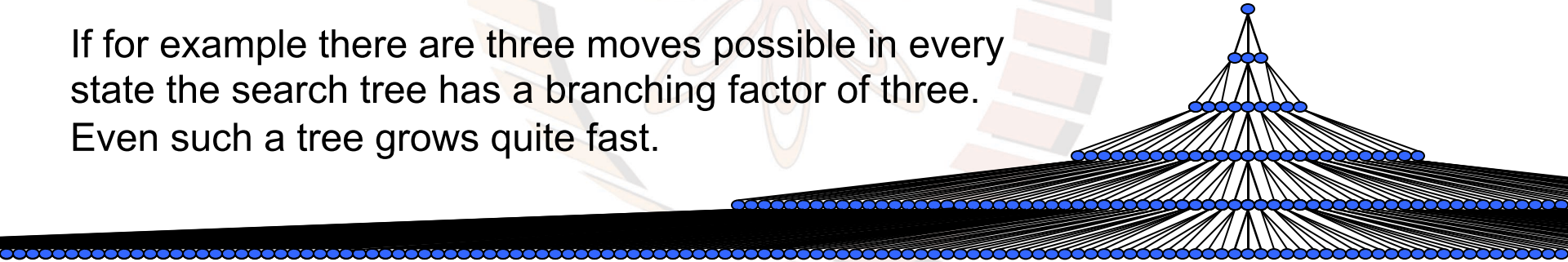
The extra cost is not significant!

CombEx

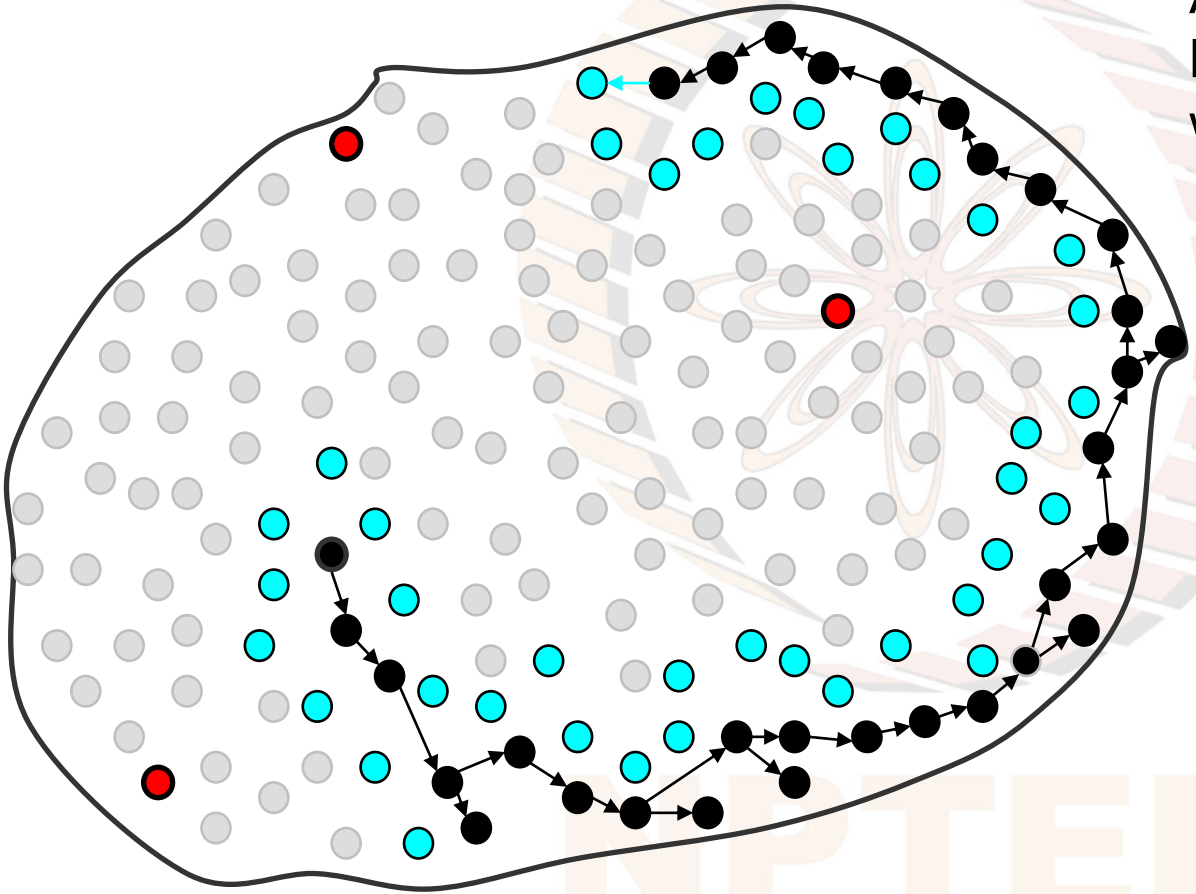
The monster that AI fights is Combinatorial Explosion.

For every node that fails the goaltest all its successors are added to the search tree.

If for example there are three moves possible in every state the search tree has a branching factor of three. Even such a tree grows quite fast.

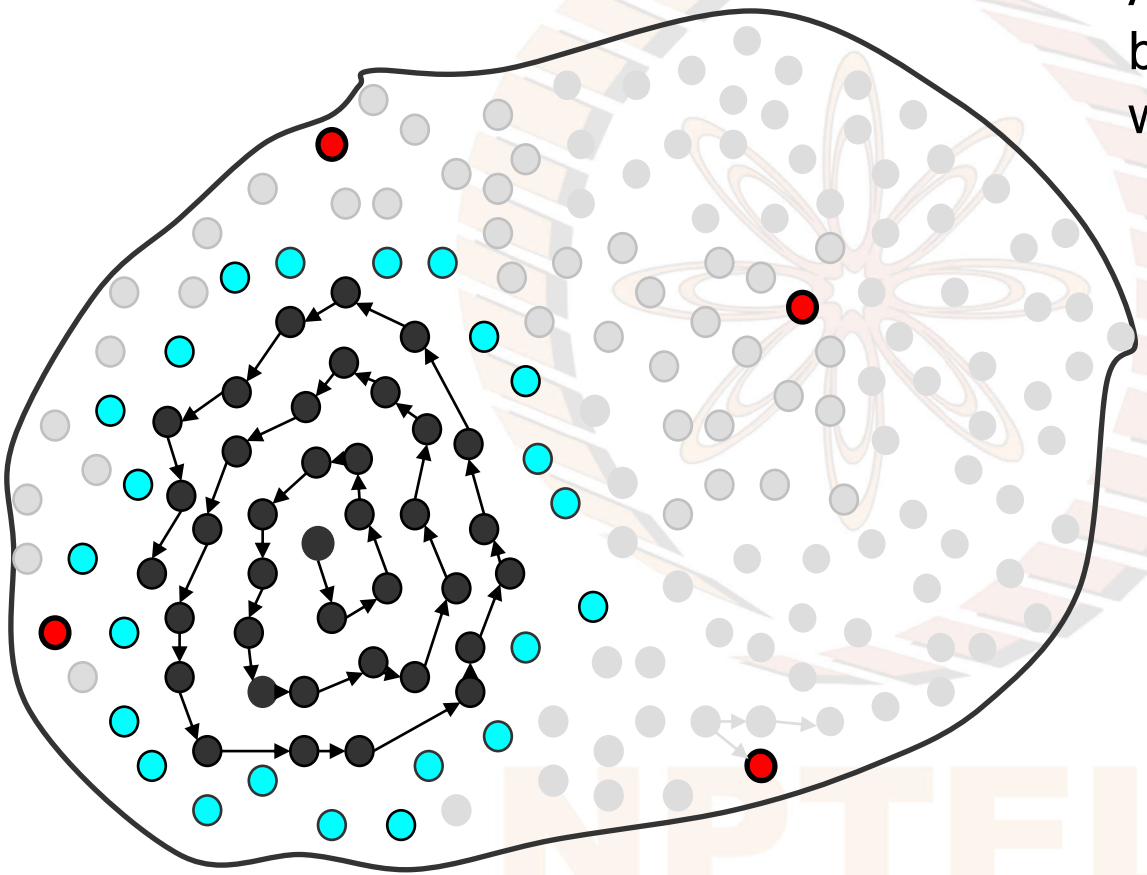


Depth First Search



Always the same behaviour irrespective of where the goal node is….

Breadth First Search



Always the same behaviour irrespective of where the goal node is….

End
Uninformed Search