# Search Methods in Artificial Intelligence

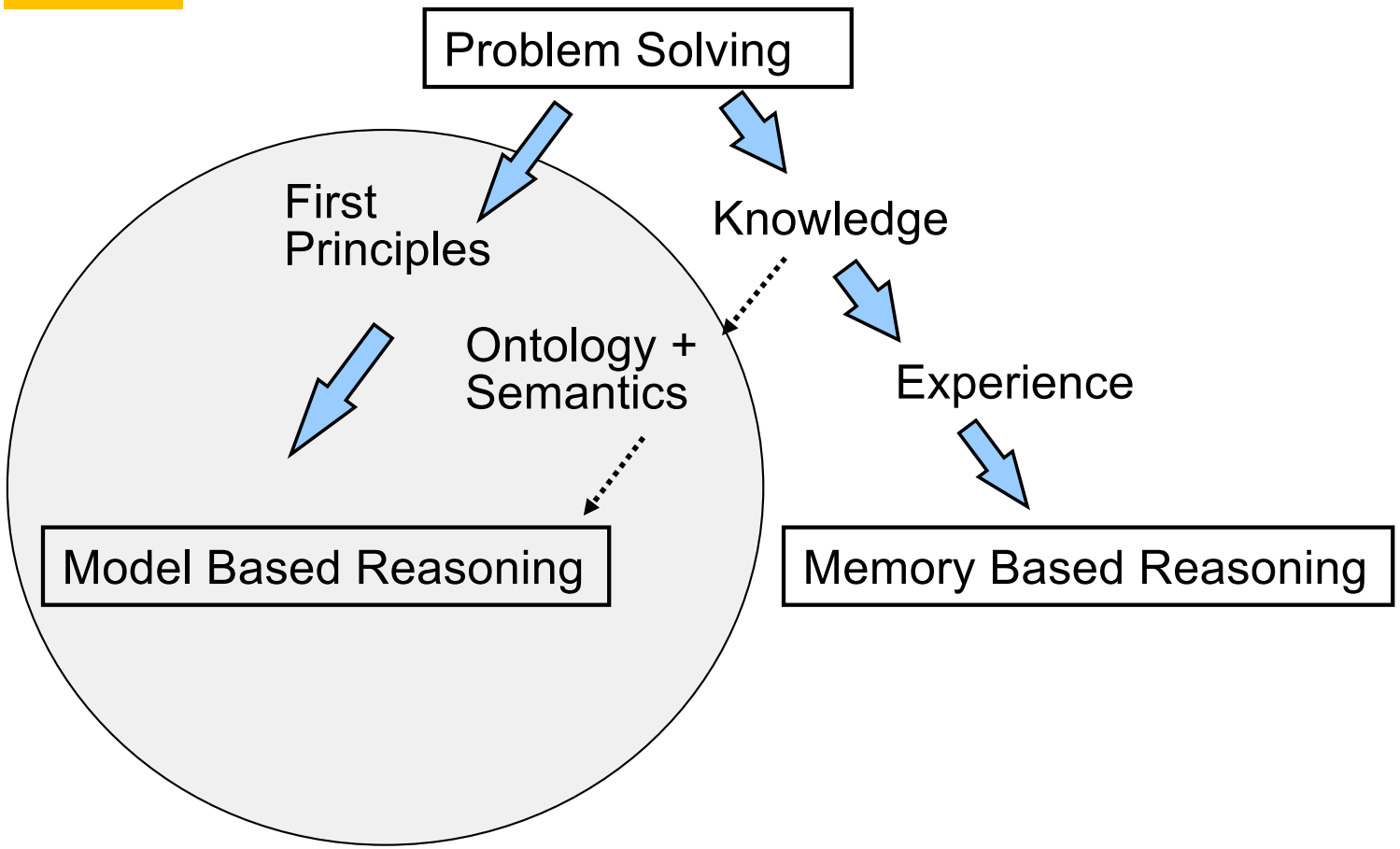
# State Space Search

Deepak Khemani
Plaksha University

# Problem Solving

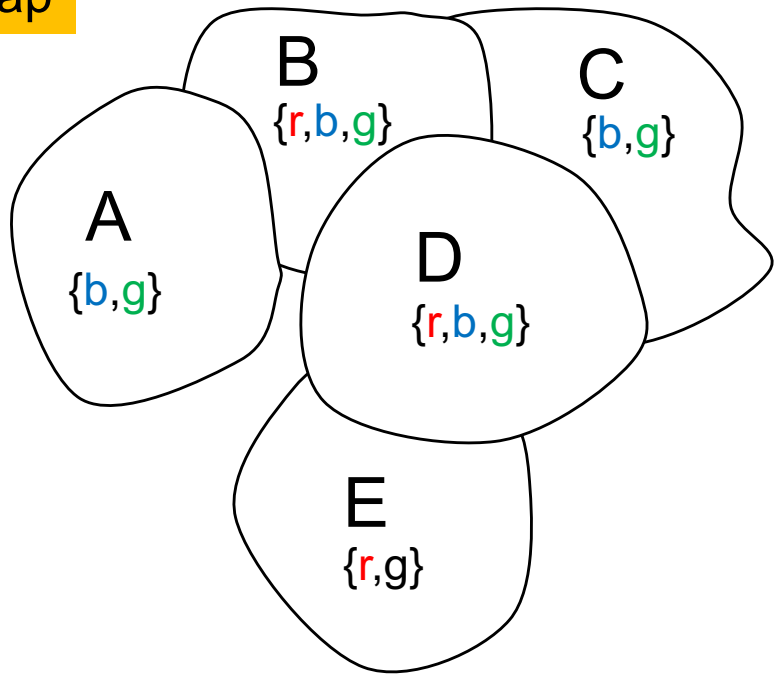
Humankind is a problem solving species.

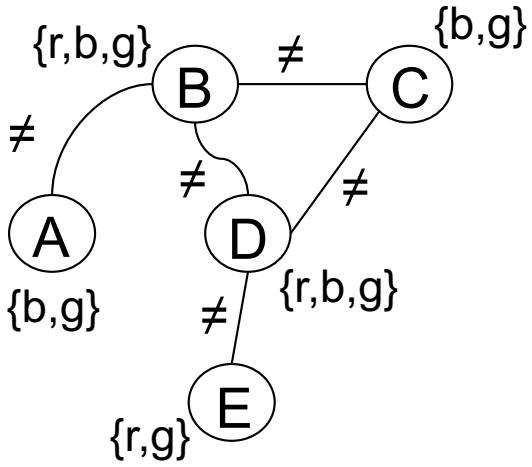
Recap



# A Map Colouring Problem

Recap




The constraint graph.
Choose a label for each node

Colour each region in the map with an allowed colour such that no two adjacent regions have the same colour.

# Solving Map Colouring

Brute force: Try all combinations of colours for all regions.

Informed Search: Choose a colour that does not conflict with neighbours.

General Search: Pose the problem to serve as an input to a general search algorithm.

- Map colouring can easily be posed as a constraint satisfaction problem. We will do so towards the end of the course.
- It can also be posed as state space search, in which the move is to assign a colour to an region in a given partially coloured state.

In this course we explore general search methods which will incorporate heuristics to go beyond brute force search.

# General Purpose Methods

Instead of writing a custom program for every problem to be solved, general purpose methods aim to write search algorithms into which individual problems can be plugged in.

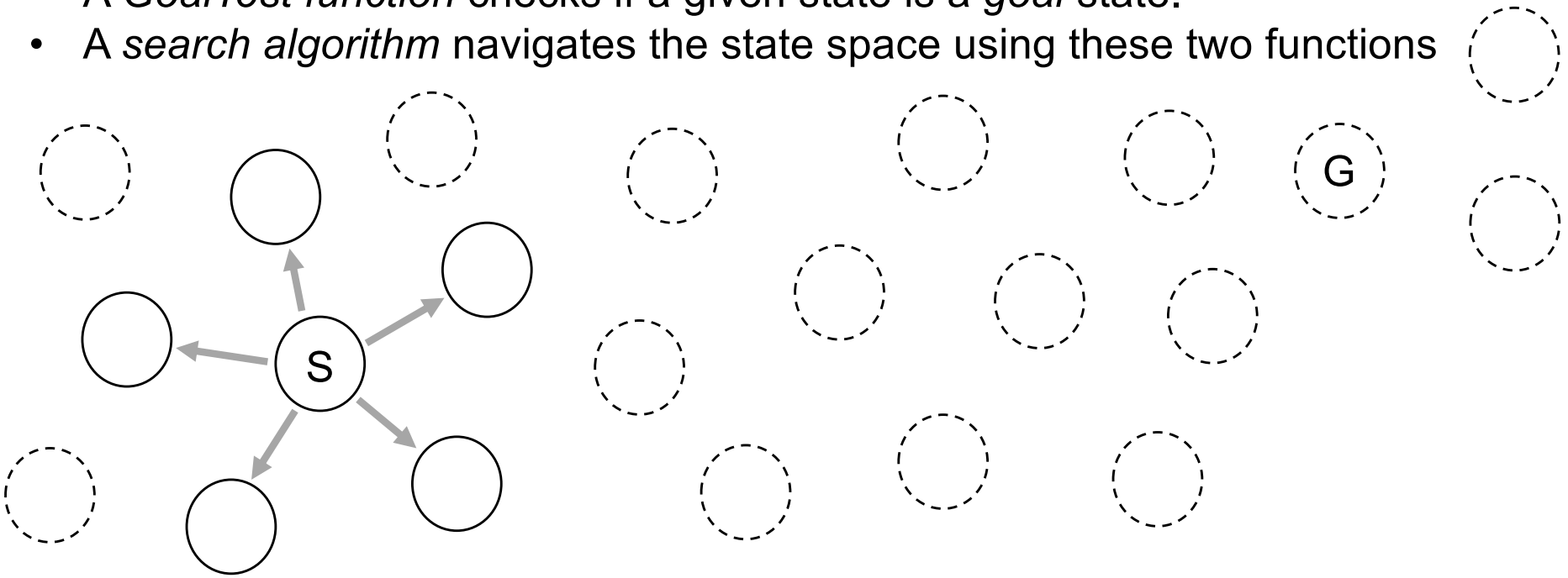
Two related approaches.

- State space search (also solution space search):
  - Describe the given state
  - Devise an operator to define the actions in each state
  - Navigate the state space in search of the desired or goal state

- Constraint Processing:
  - Define a set of variables each with their own domain
  - Describe the constraints between subsets of variables
  - Search for an assignment for each variable such that all constraints are satisfied

# State Space Search → Graph Search

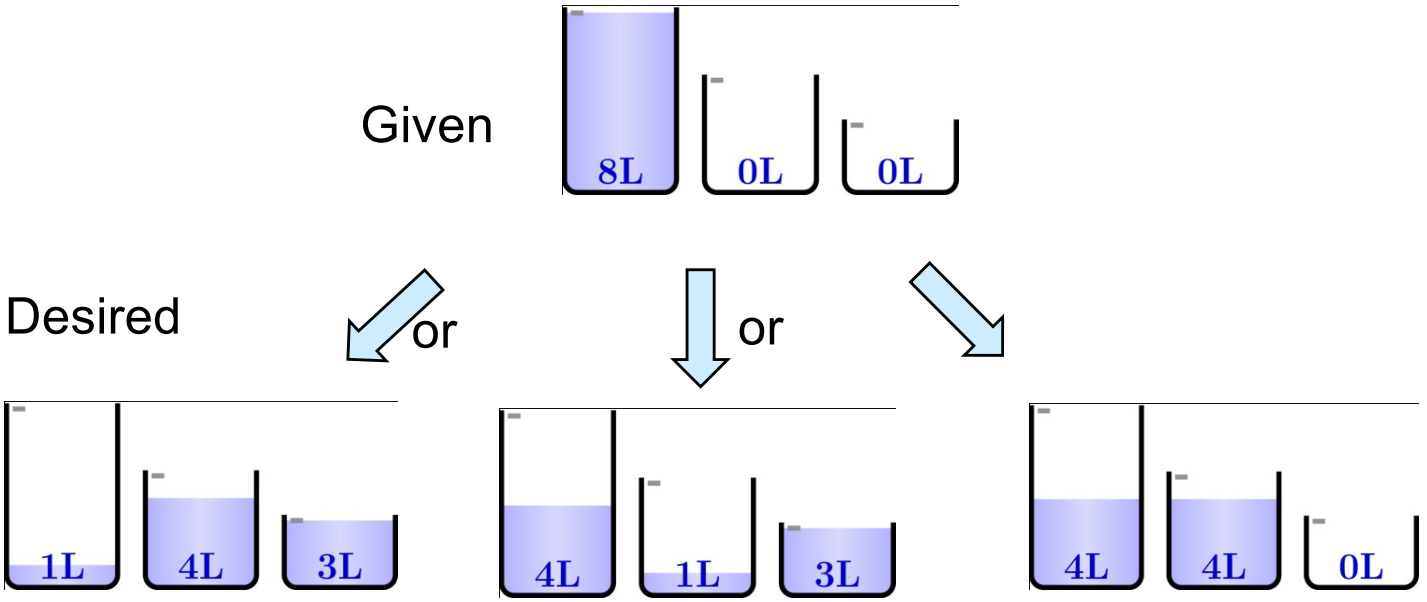
- A state is a representation of a situation.
- A *MoveGen function* captures the moves that can be made in a given state.
- It returns a set of states – *neighbours* – resulting from the moves.
- The state space is an *implicit* graph defined by the MoveGen function.
- A *GoalTest function* checks if a given state is a *goal* state.
- A *search algorithm* navigates the state space using these two functions



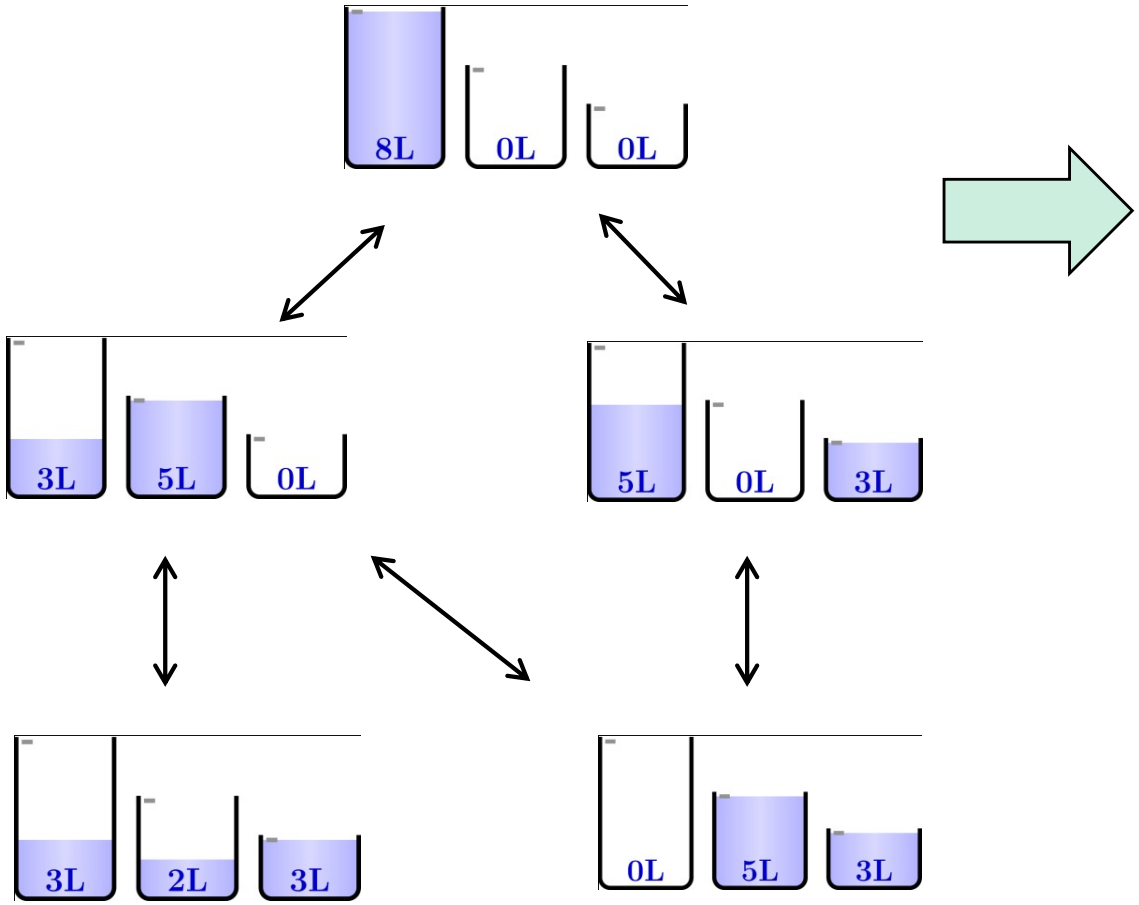
# Some sample problems

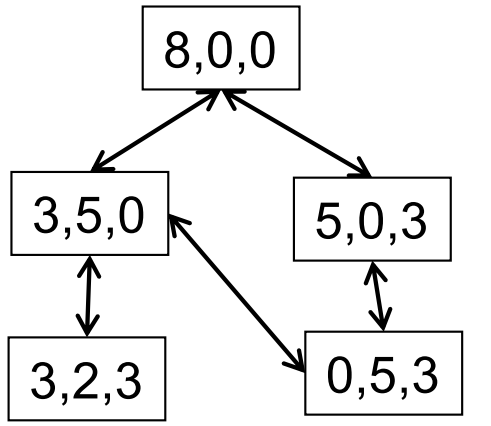
# The Water Jug problem

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



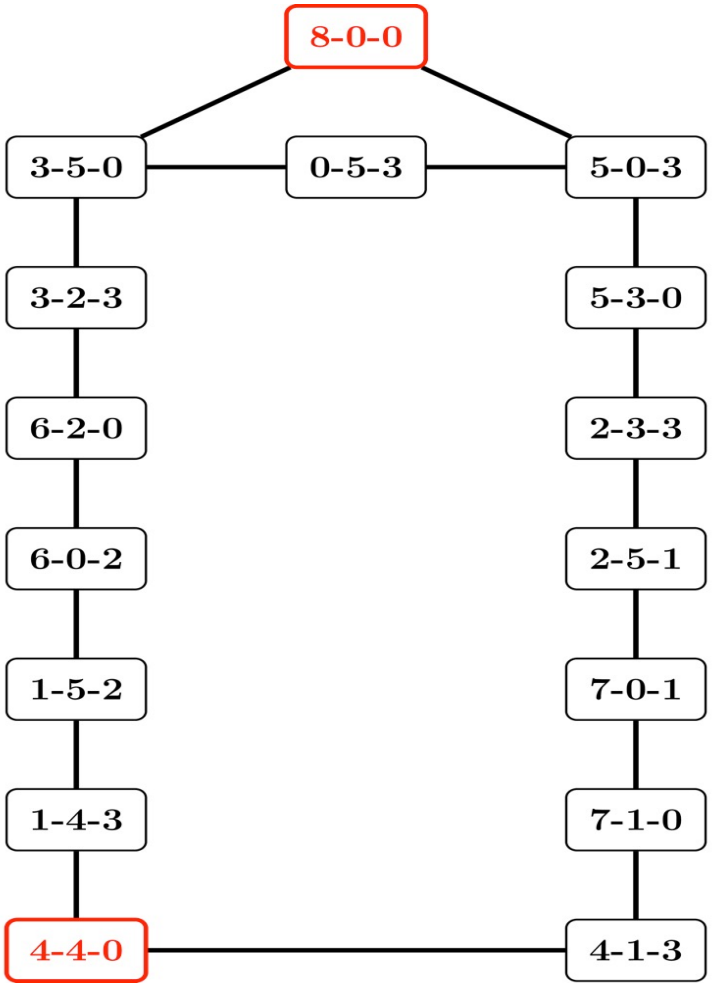
# Water Jug Problem : Some Sample Moves




State transition graph

# The Solution : (8,0,0) → (4,4,0)




Various solutions (paths) for reaching the state (4,4,0) exist.

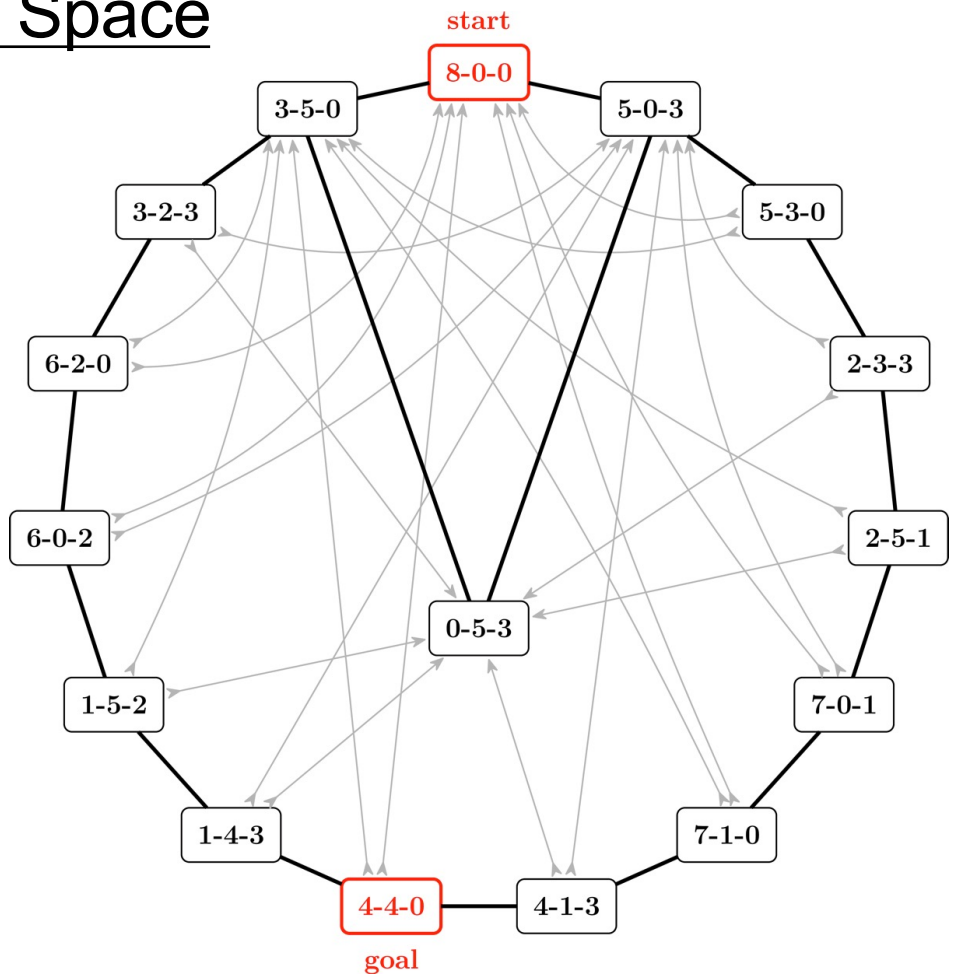
Exercise: Write a *GoalTest* function to return *true* when any jug has 4 liters.

We do not consider those moves where one can throw away some water.

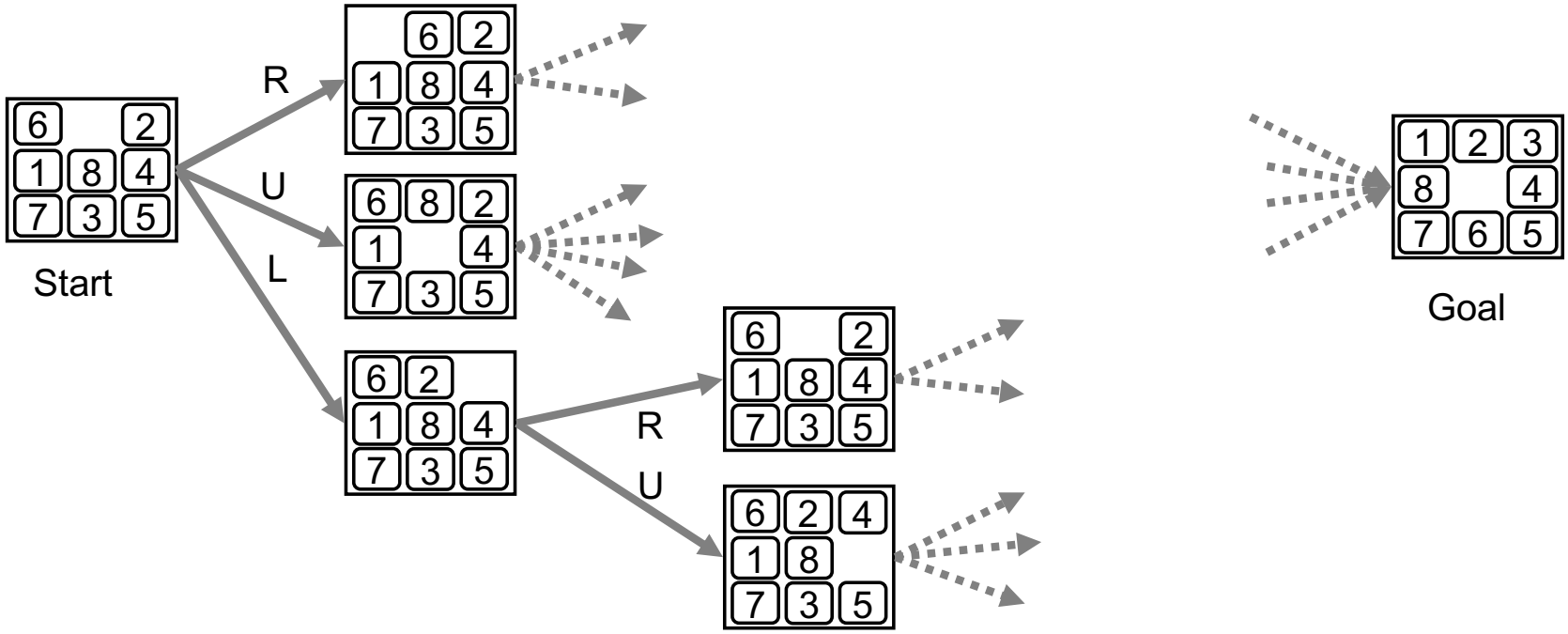
# The Complete State Space

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

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



# The Eight-puzzle



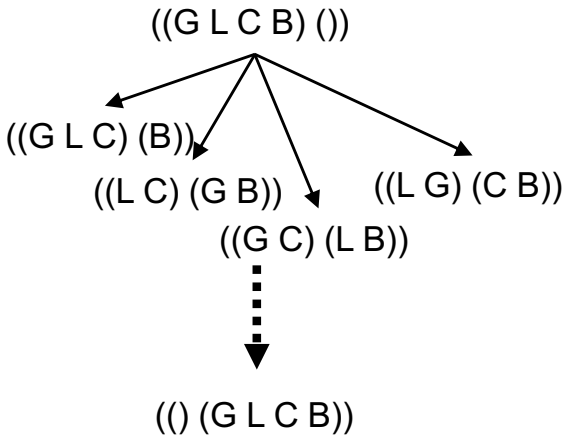

The Eight puzzle consists of eight tiles on a 3x3 grid. A tile can slide into an adjacent location if it is empty. A move is labeled R if a tile moves right, and likewise for up (U), down (D) and left (L). *Planning* algorithms use *named* moves.

# 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

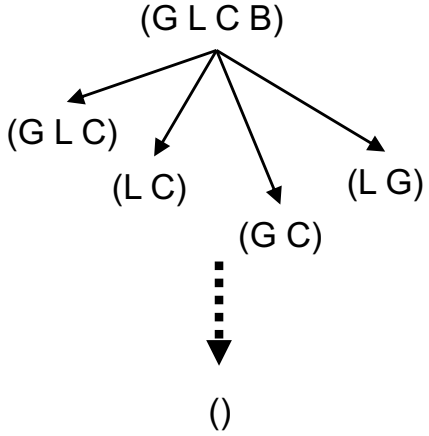
Exercise:
Write the *MoveGen* function
Write the *GoalTest* function

# 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

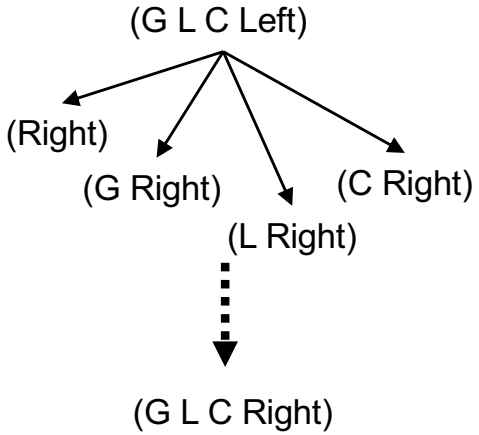
Exercise:
Write the *MoveGen* function
Write the *GoalTest* function

# 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?

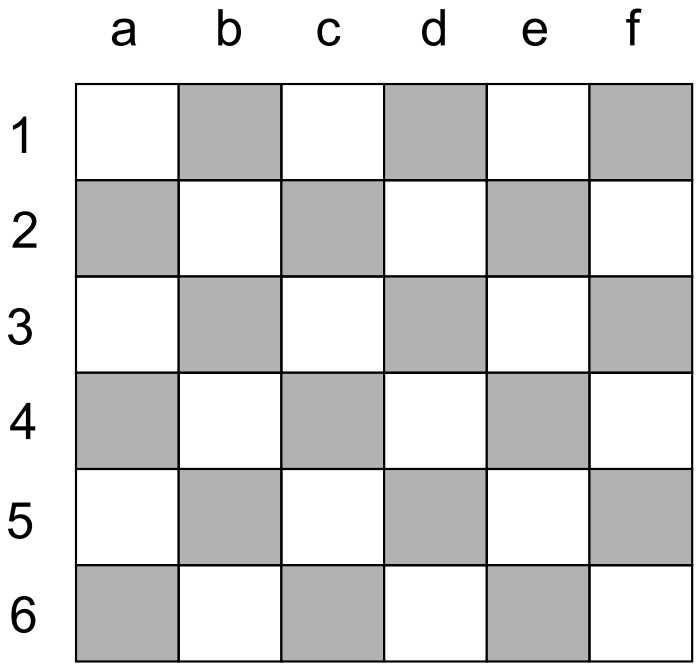
Objects on the side where the boat is

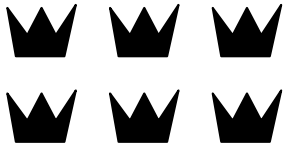



Man M is in each state shown

Exercise:
Write the *MoveGen* function
Write the *GoalTest* function

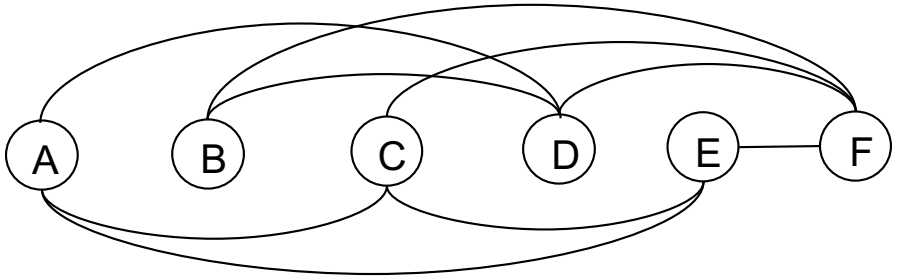
# The N-queens problem



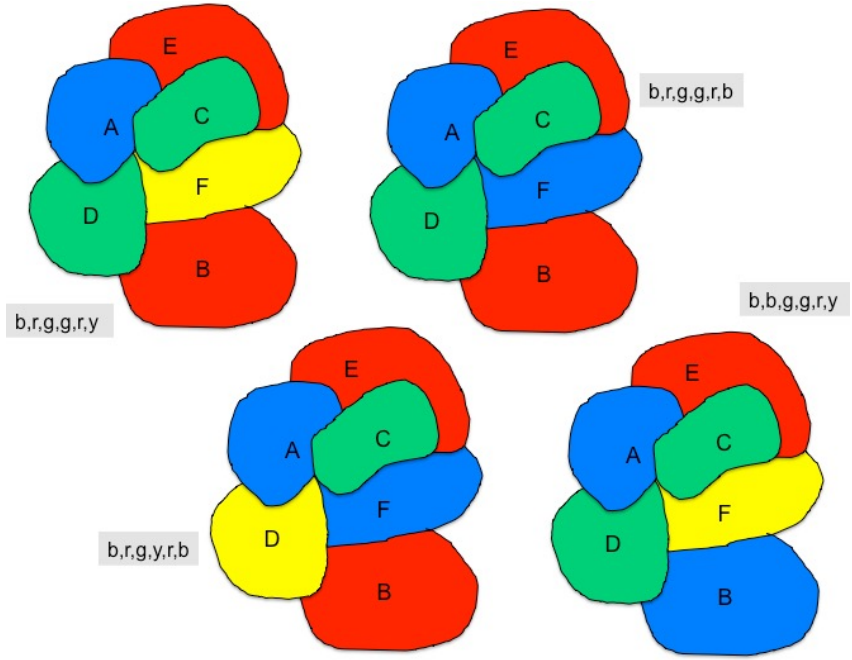



The 6-queen problem is to place the six queens on a 6x6 chess board such that no queen attacks another.

# A map colouring problem and its solutions

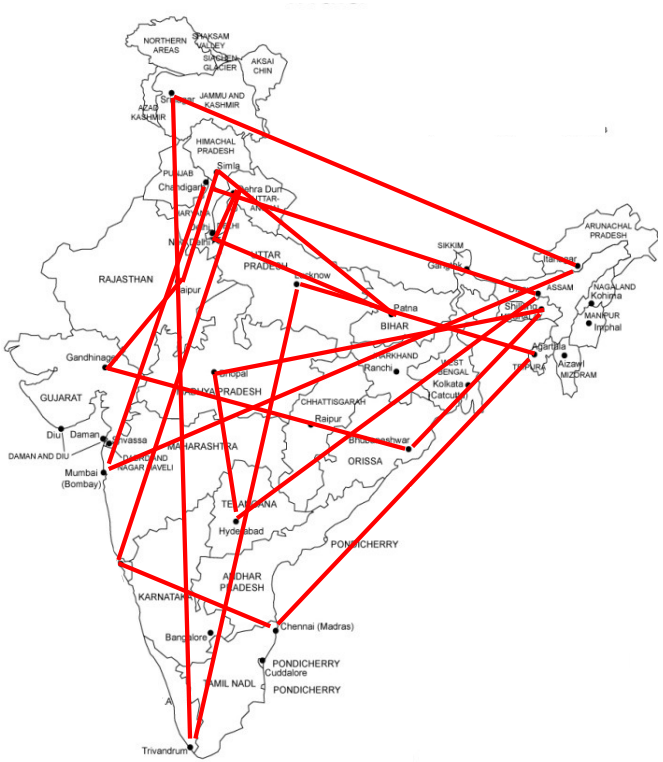



{b,g} {b,r} {g} {g,y} {r,g} {b,y}

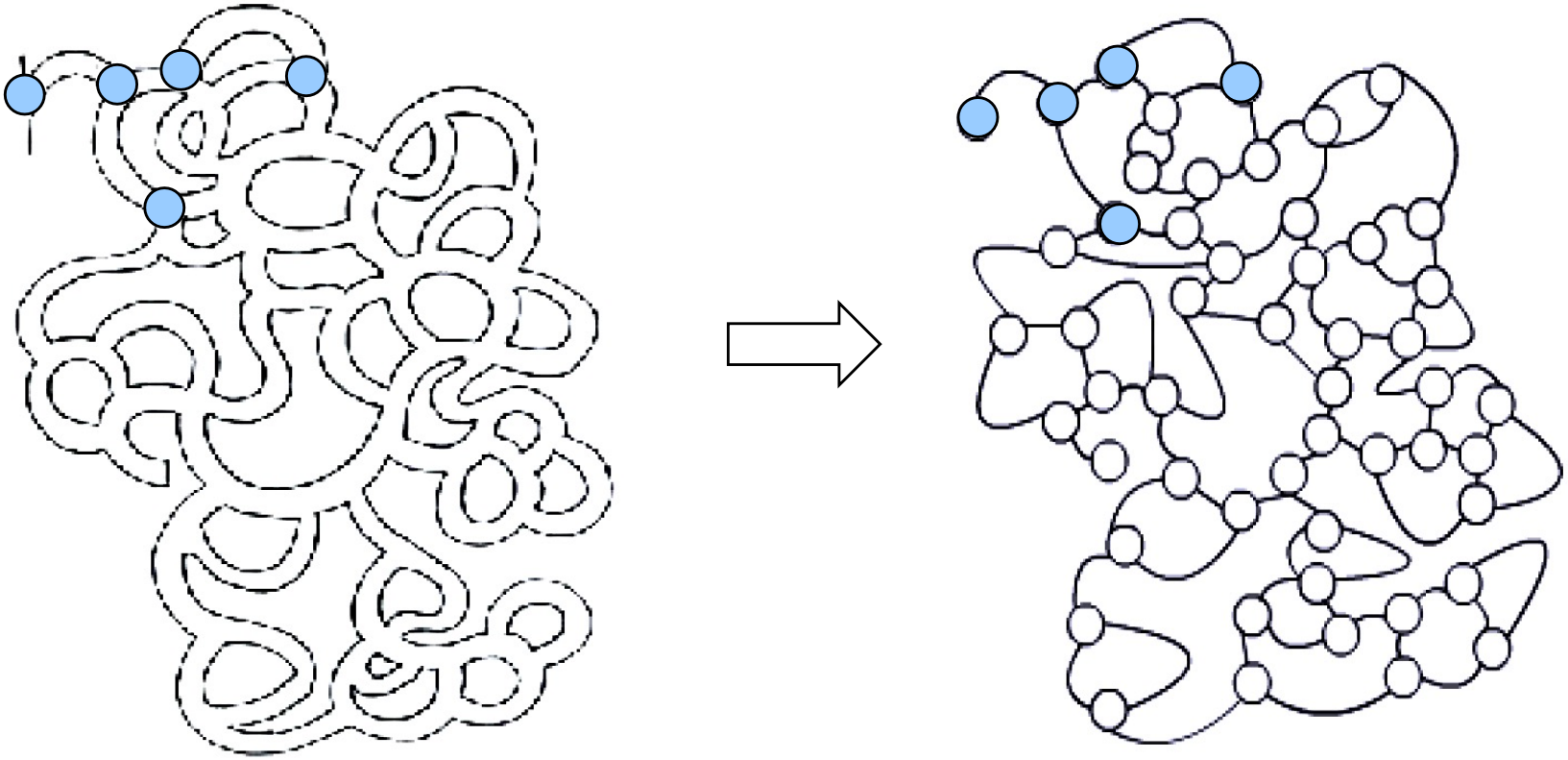



The order represents the order in which the regions A, B, C, D, E, and F are assigned colours by the algorithm.

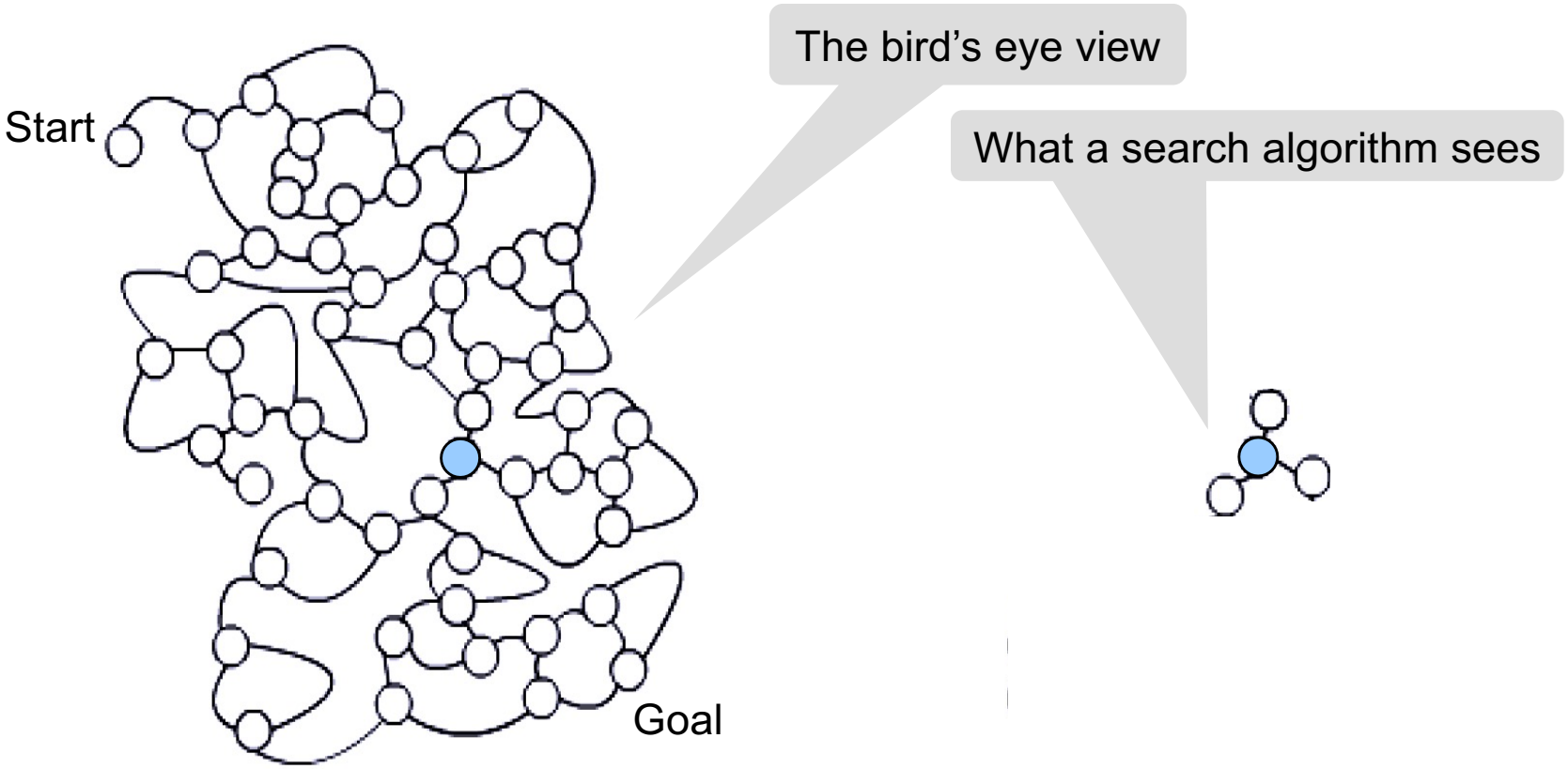
# The Traveling Salesman Problem – The Holy Grail of Computer Science




# A Maze



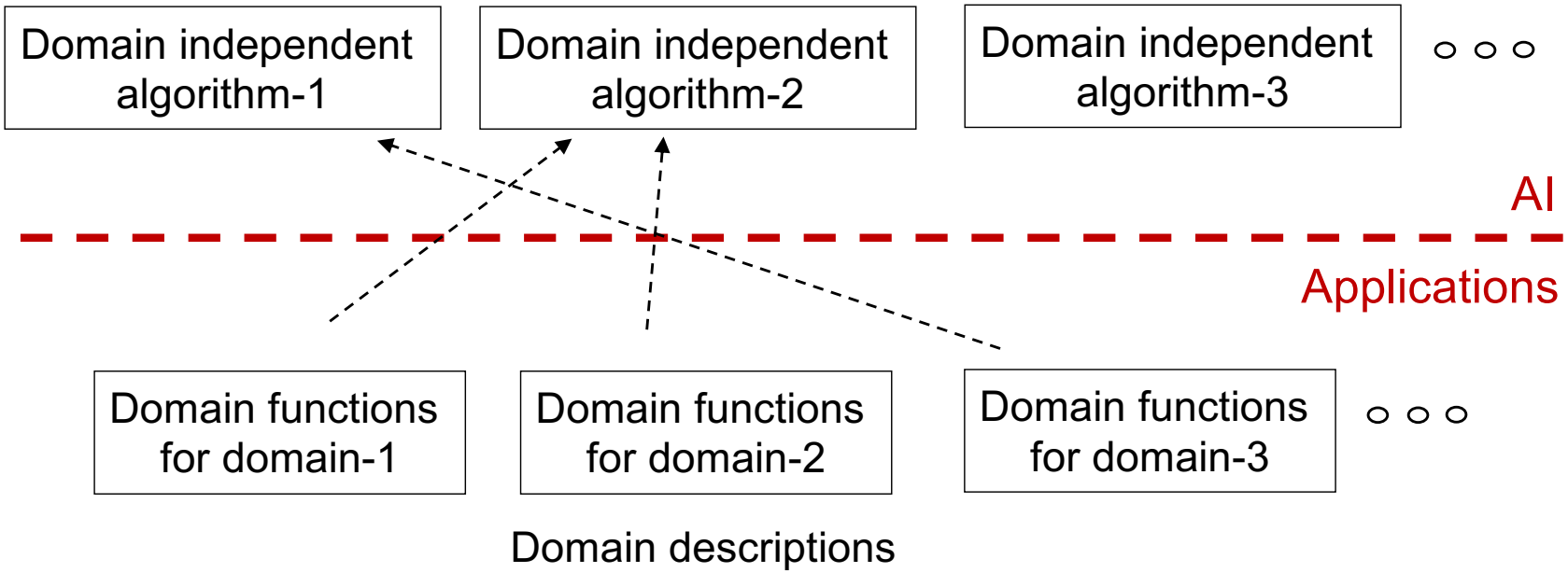
# Path finding in a maze – graph search



# General Search Algorithms

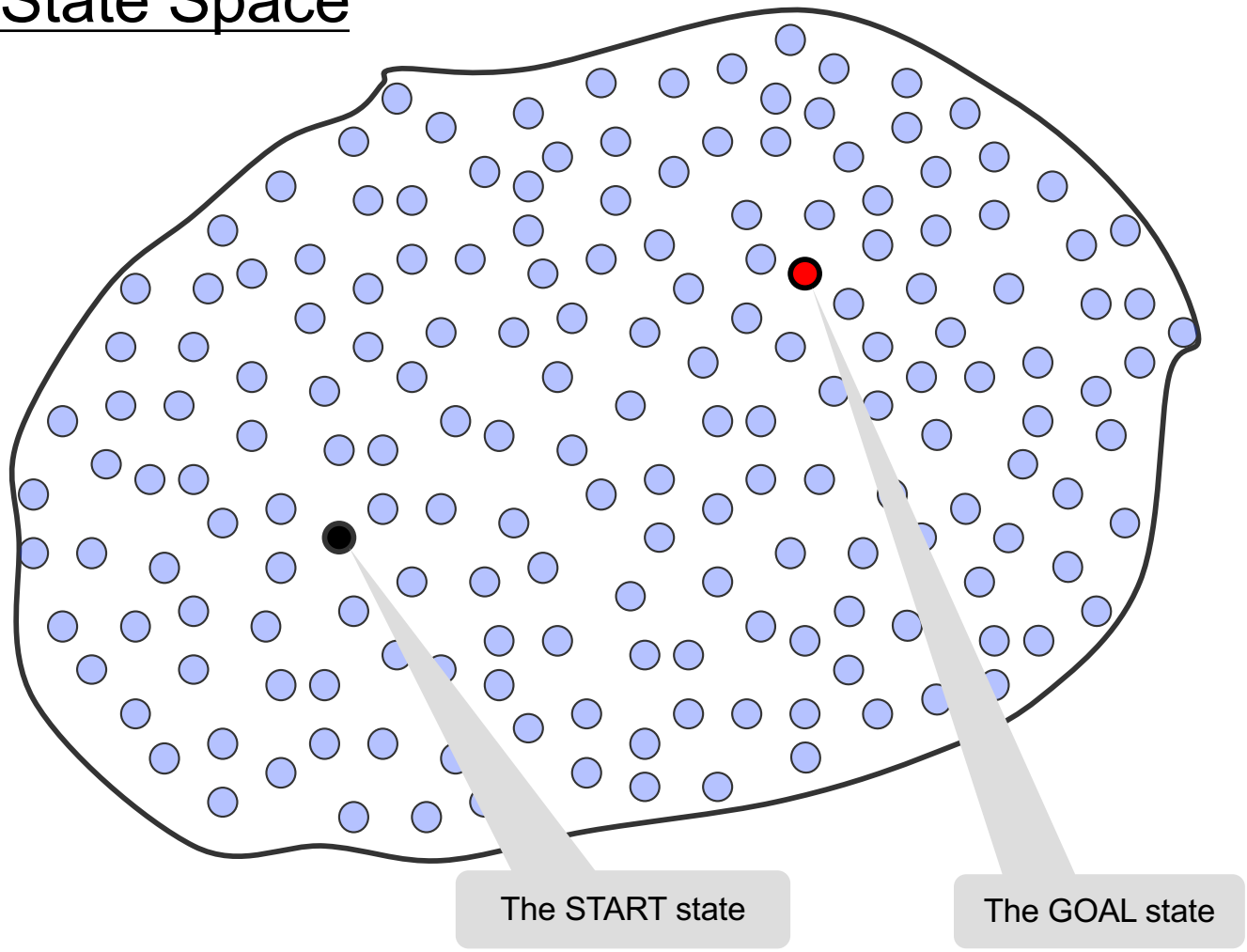
## Domain independent problem solving algorithms

Develop general problem solving algorithms in a domain independent form.

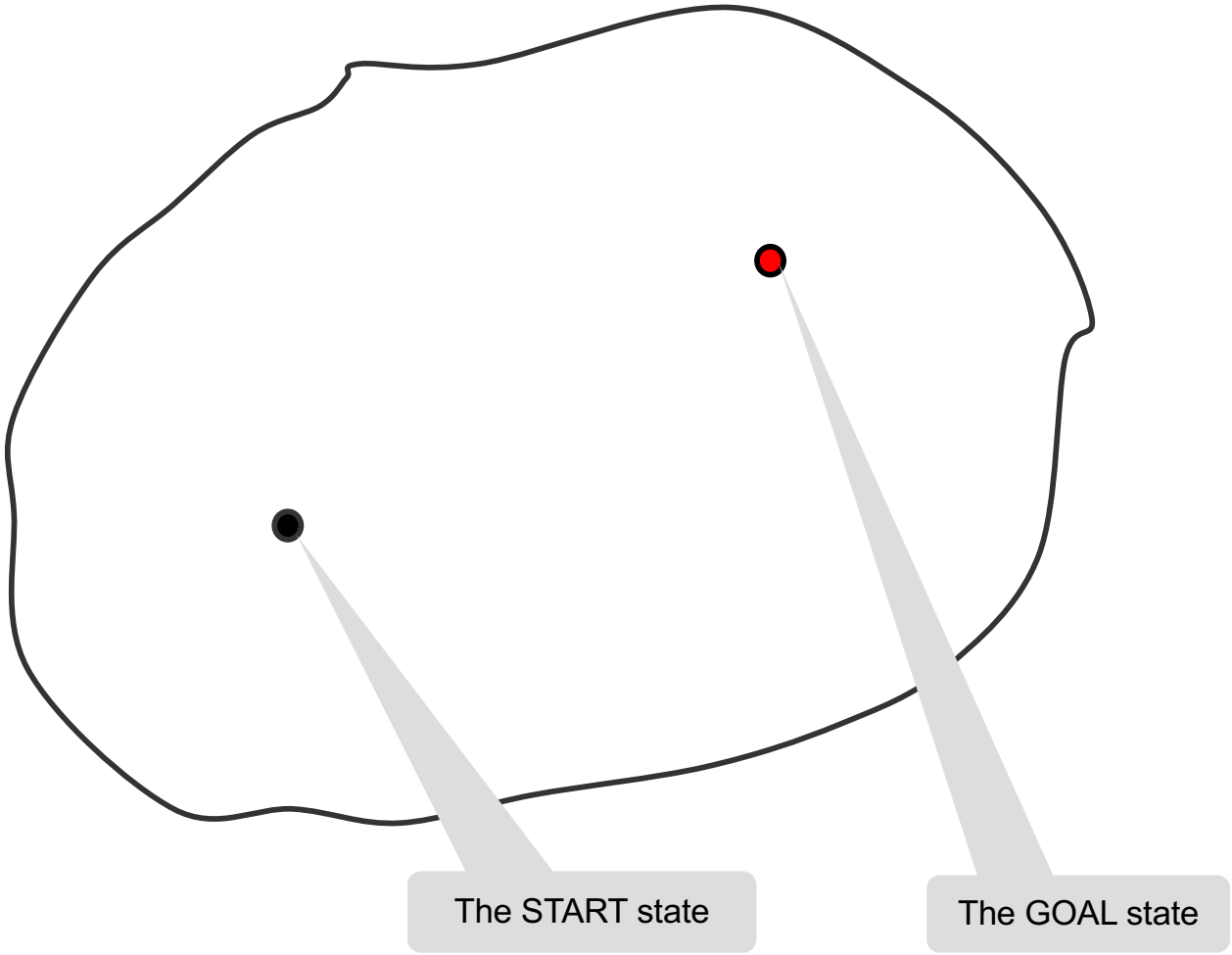



For a domain the user needs to create a domain description and plug it in a suitable problem solving algorithm.

# The State Space

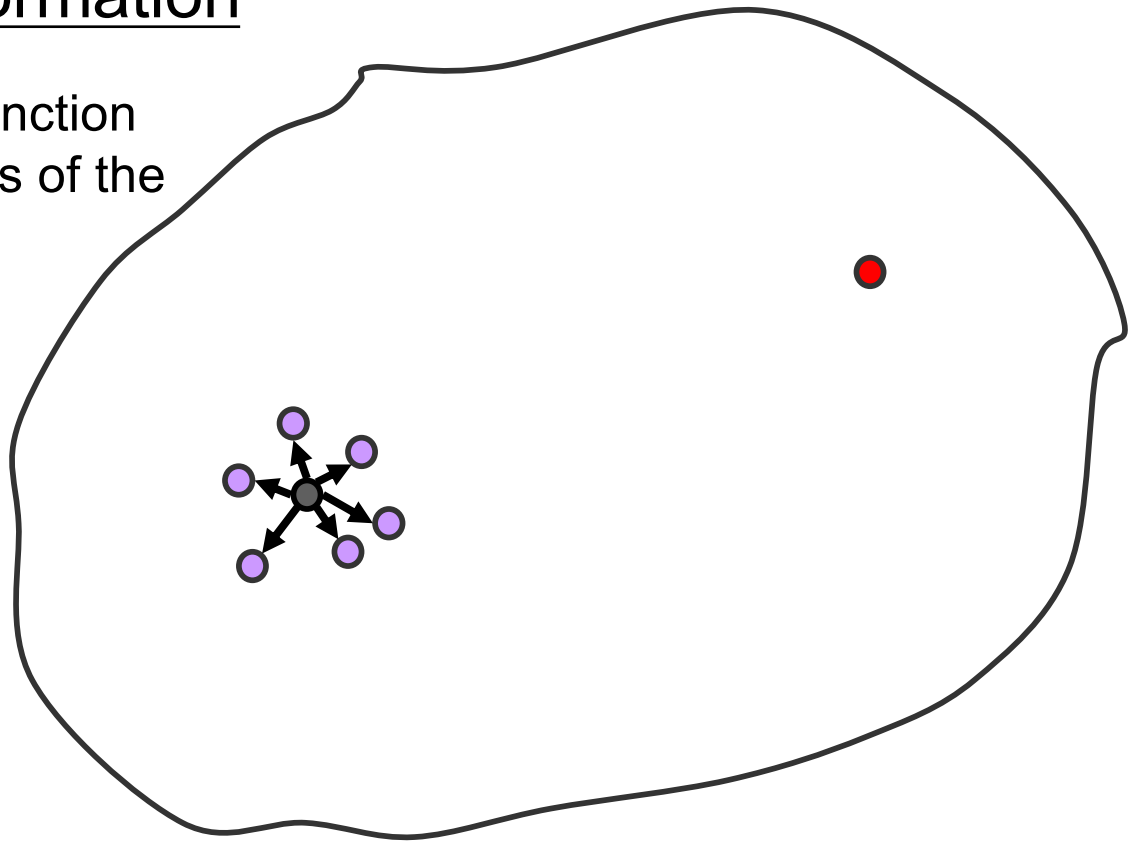


# The State Space is Implicit

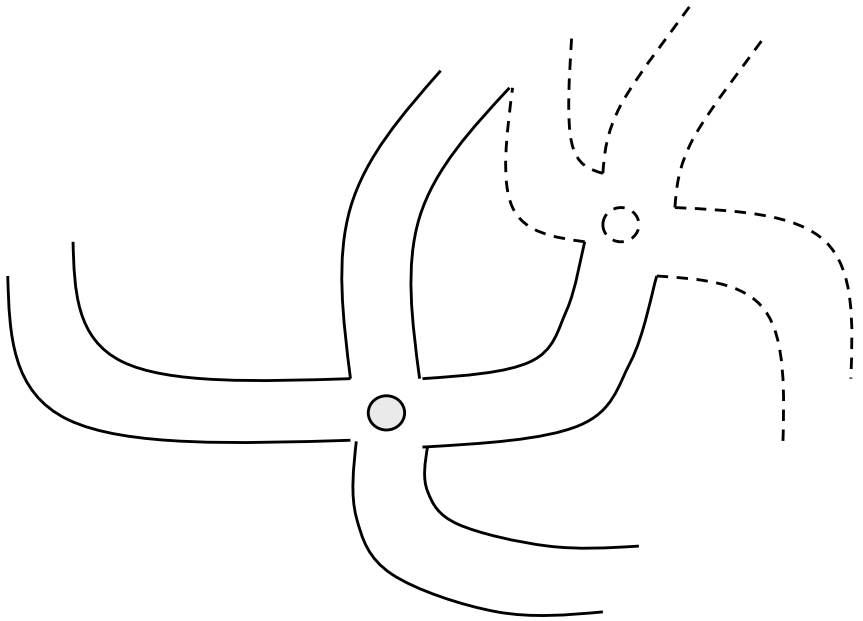


# Moves : State Transformation

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



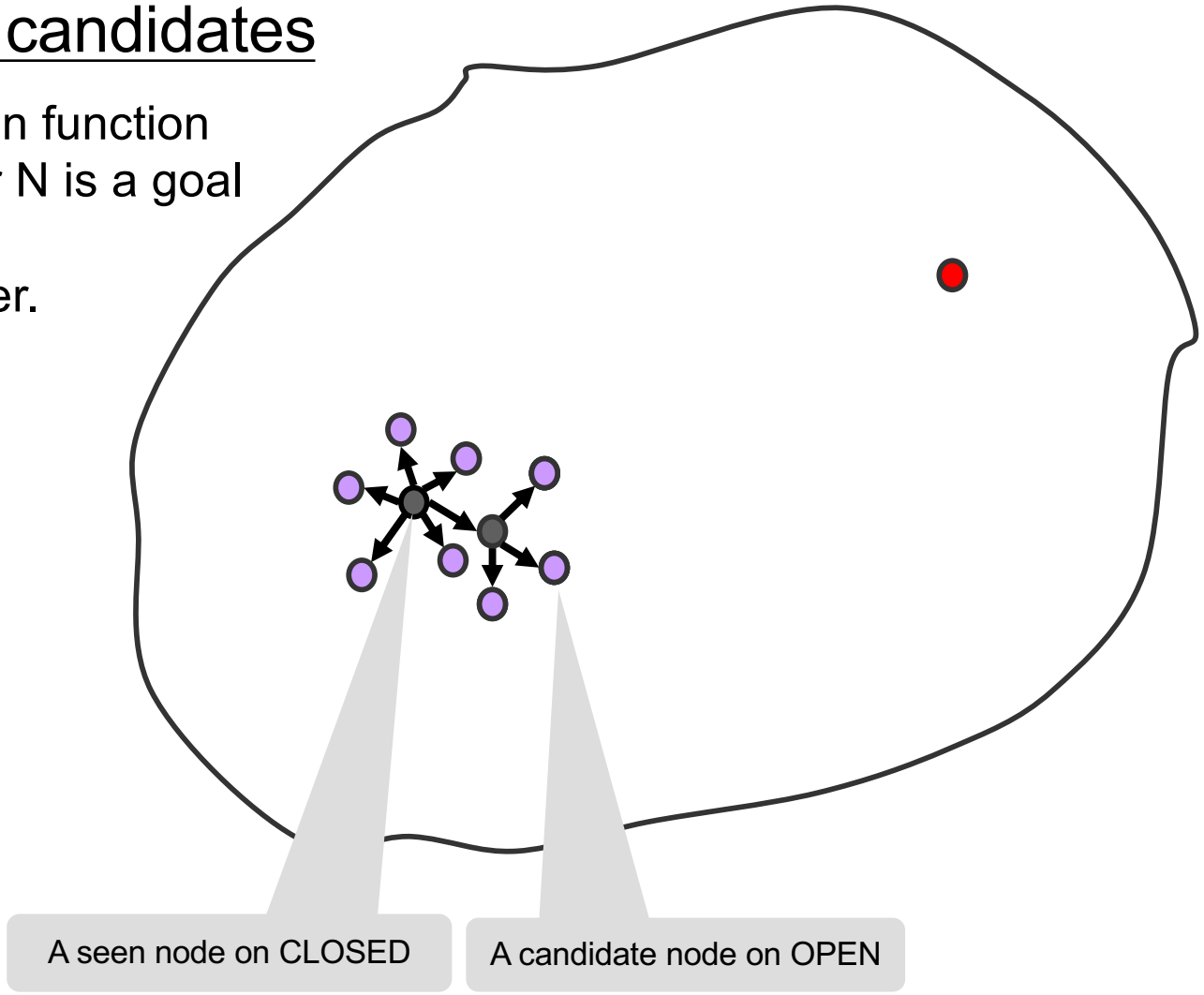
# A search node is like a junction in a maze



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

# 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.



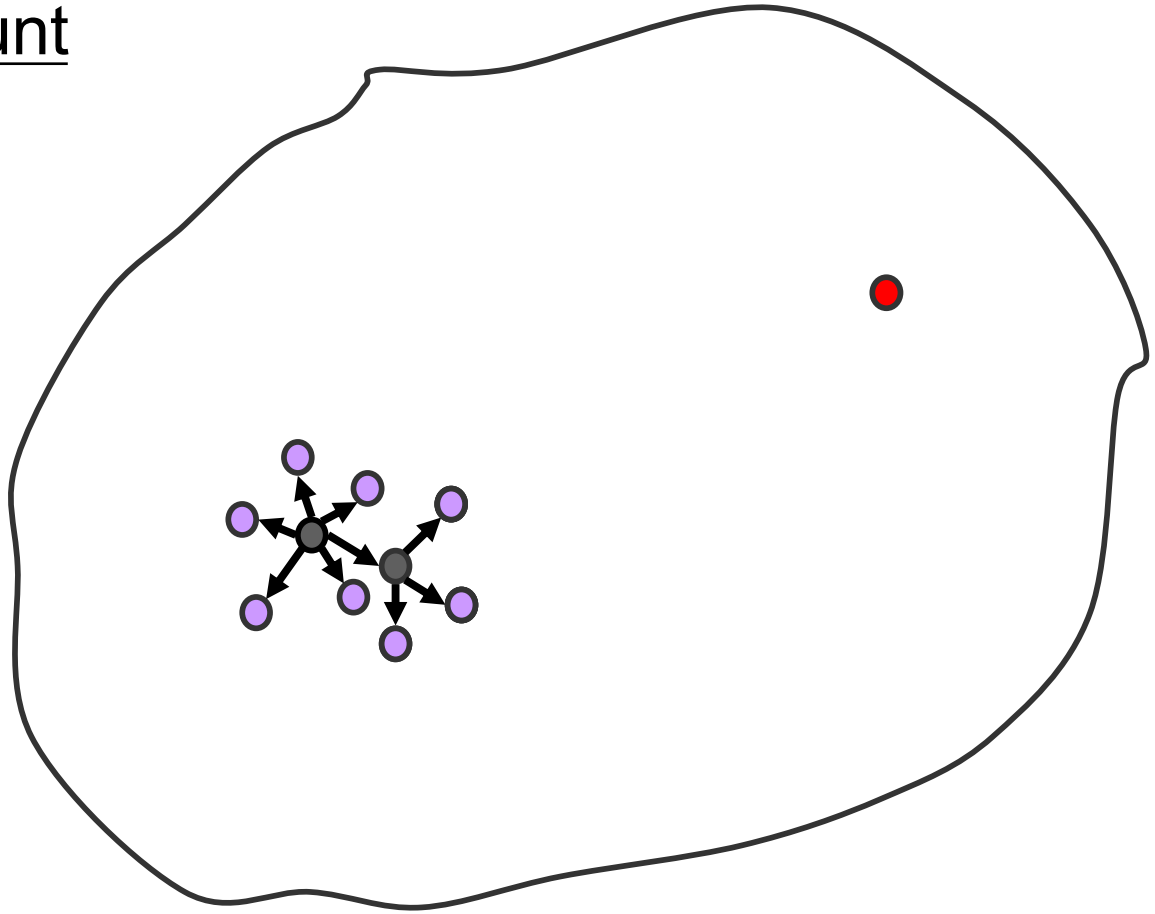
# Search = Treasure Hunt

Generate & Test :

Traverse the space by generating new nodes (using *MoveGen*)

and

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



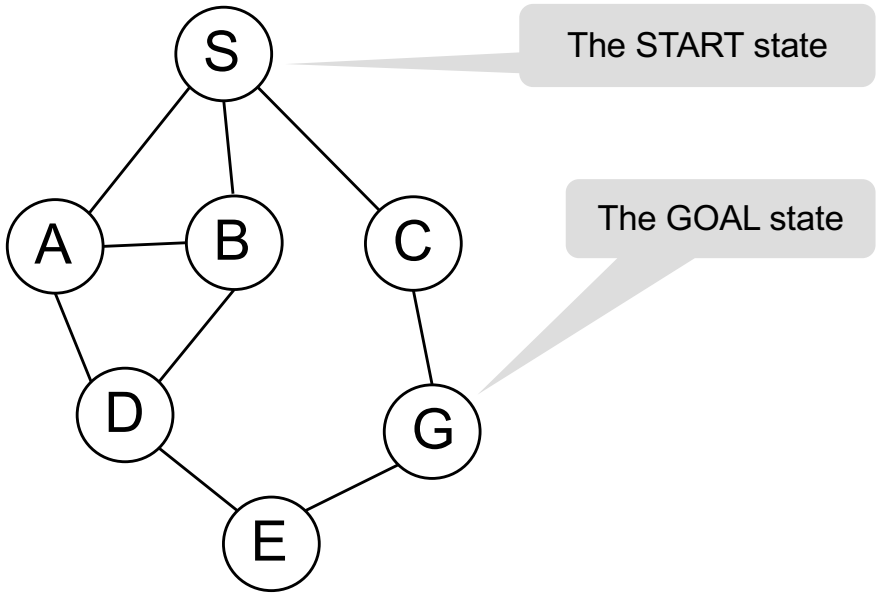
Key question : *Which* node to pick from OPEN?!

# 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

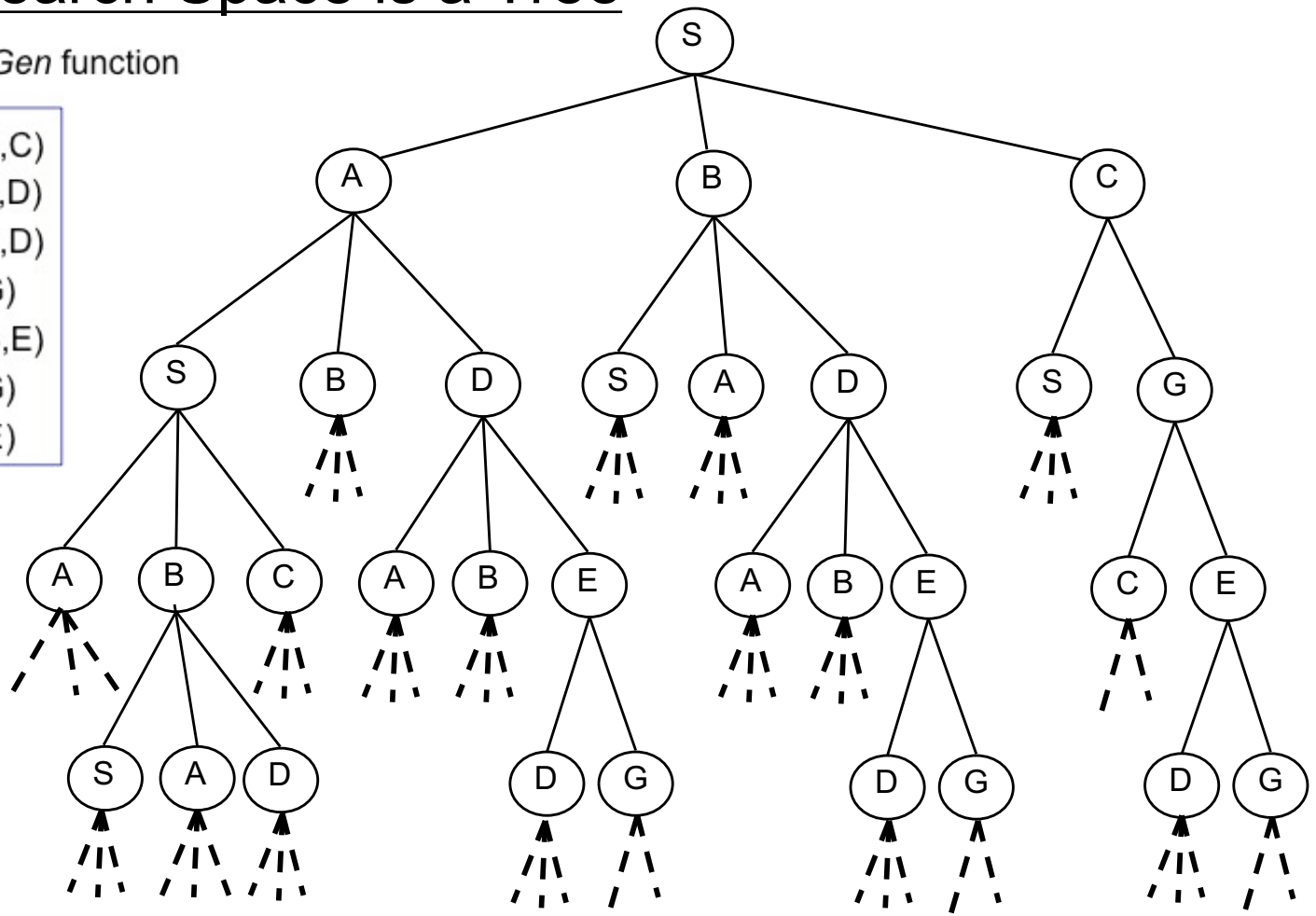



We have represented the *set* of neighbours as a *list* returned by the *MoveGen* function

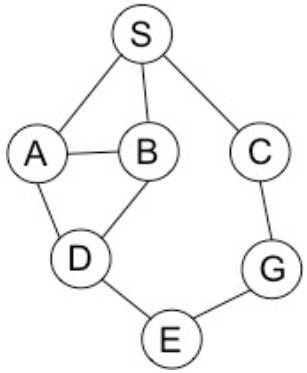
# The Search Space is a Tree

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

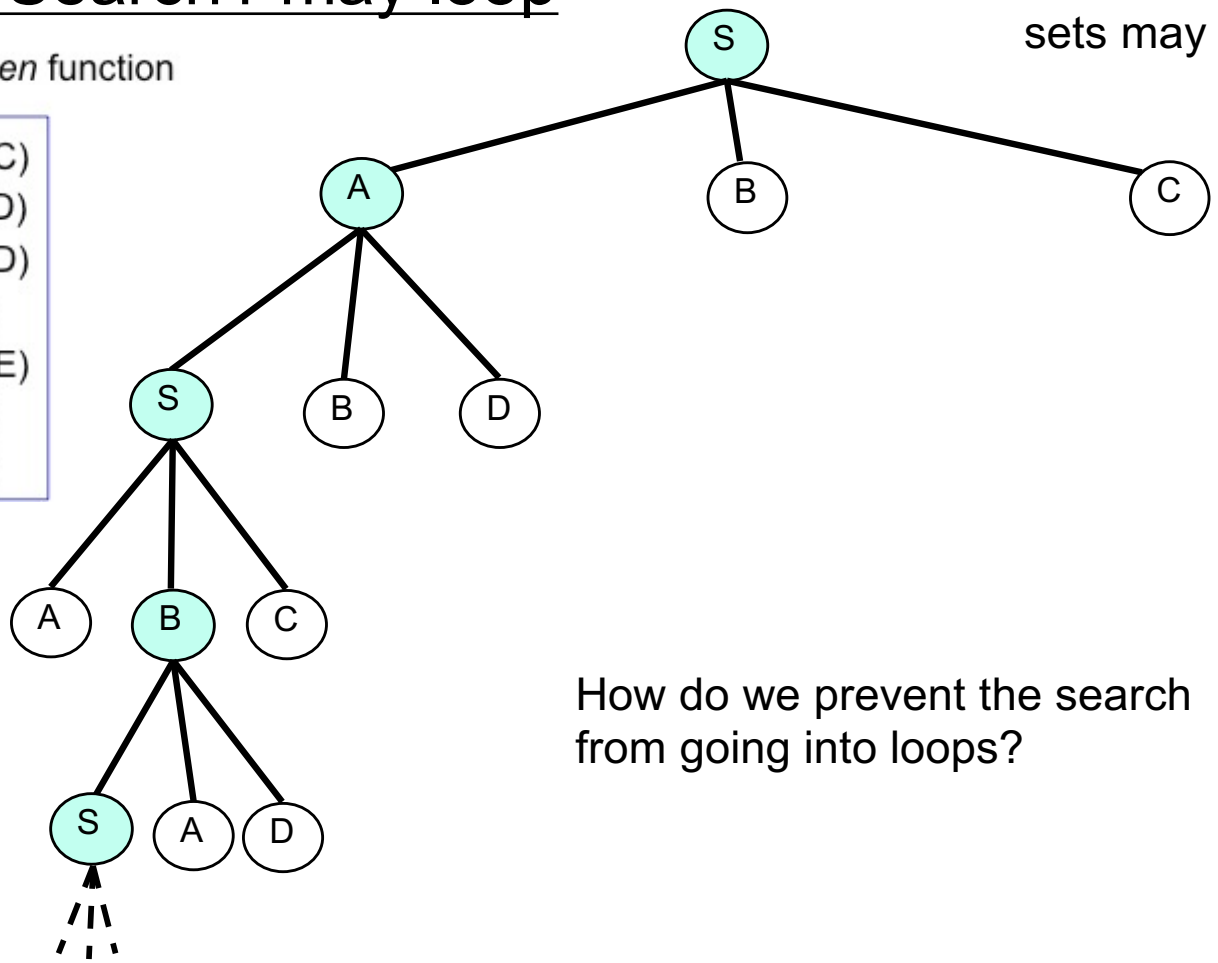
```
SimpleSearch1()
    1    OPEN ← {S}
    2    while OPEN is not empty
    3      do pick some node N from OPEN
    4              OPEN ← OPEN – {N}
    5              if GoalTest(N) = TRUE
    6                then    return N
    7                else    OPEN ← OPEN ∪ MoveGen(N)
    8    return FAILURE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

Algorithm *SimpleSearch1*

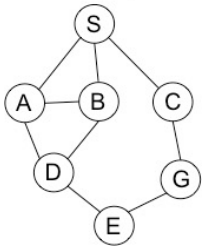
# SimpleSearch1 may loop

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)

How do we prevent the search from going into loops?

# 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
```

When a node is picked from OPEN, add it to CLOSED

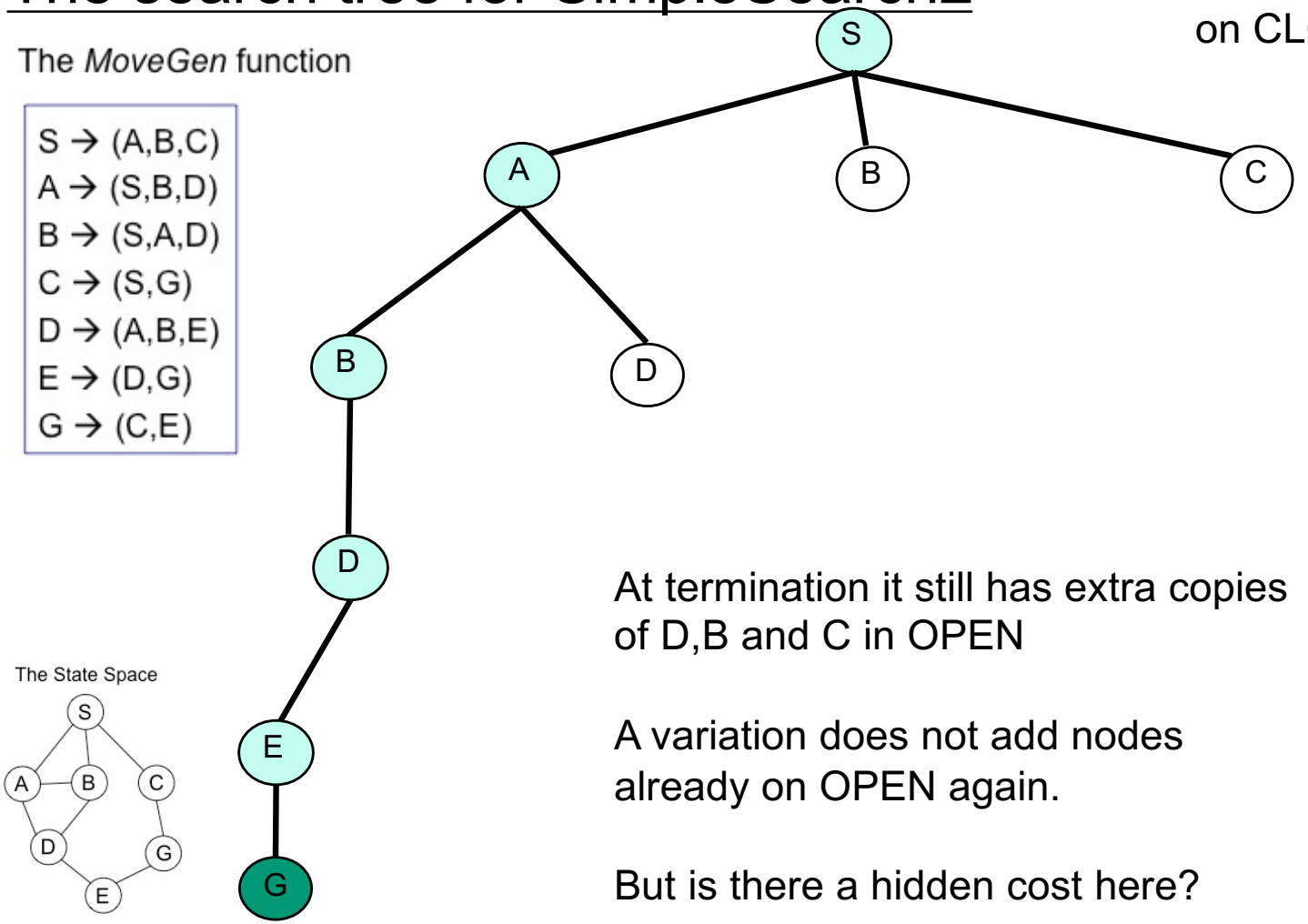
If a new node is on CLOSED do not add it to OPEN

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.

But is there a hidden cost here?

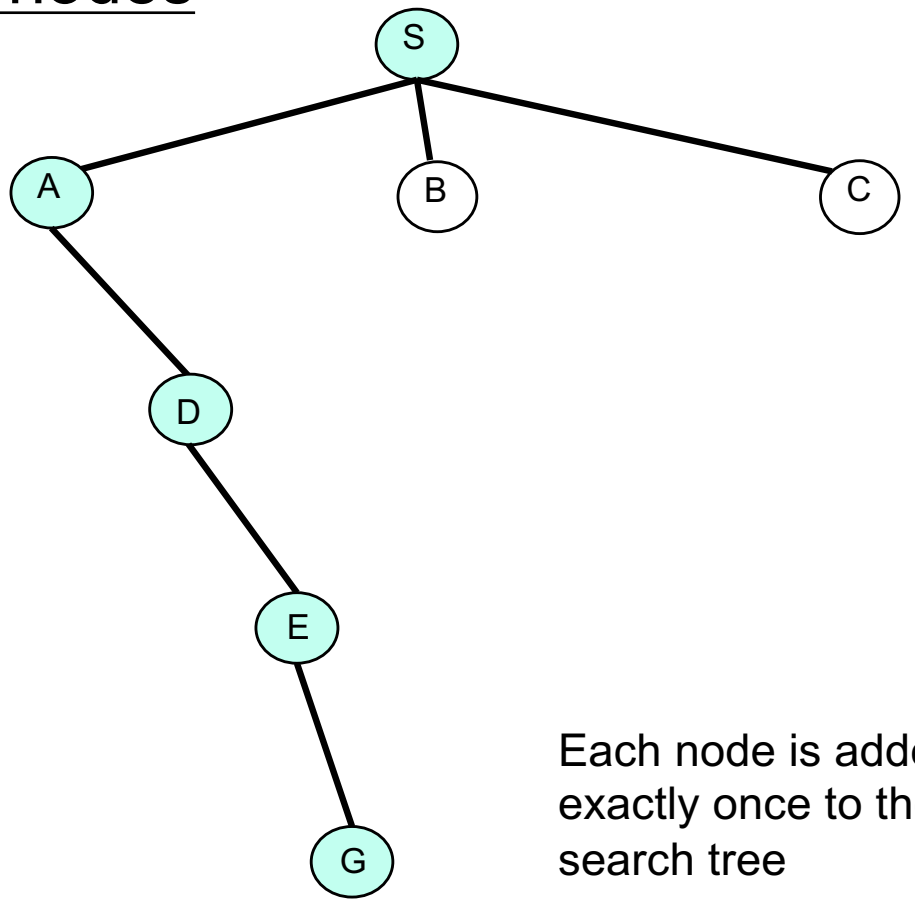
It does not add nodes on CLOSED to OPEN

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

# Adding *only* new nodes

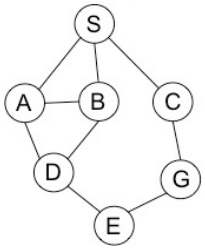
The *MoveGen* function

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)




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

The State Space

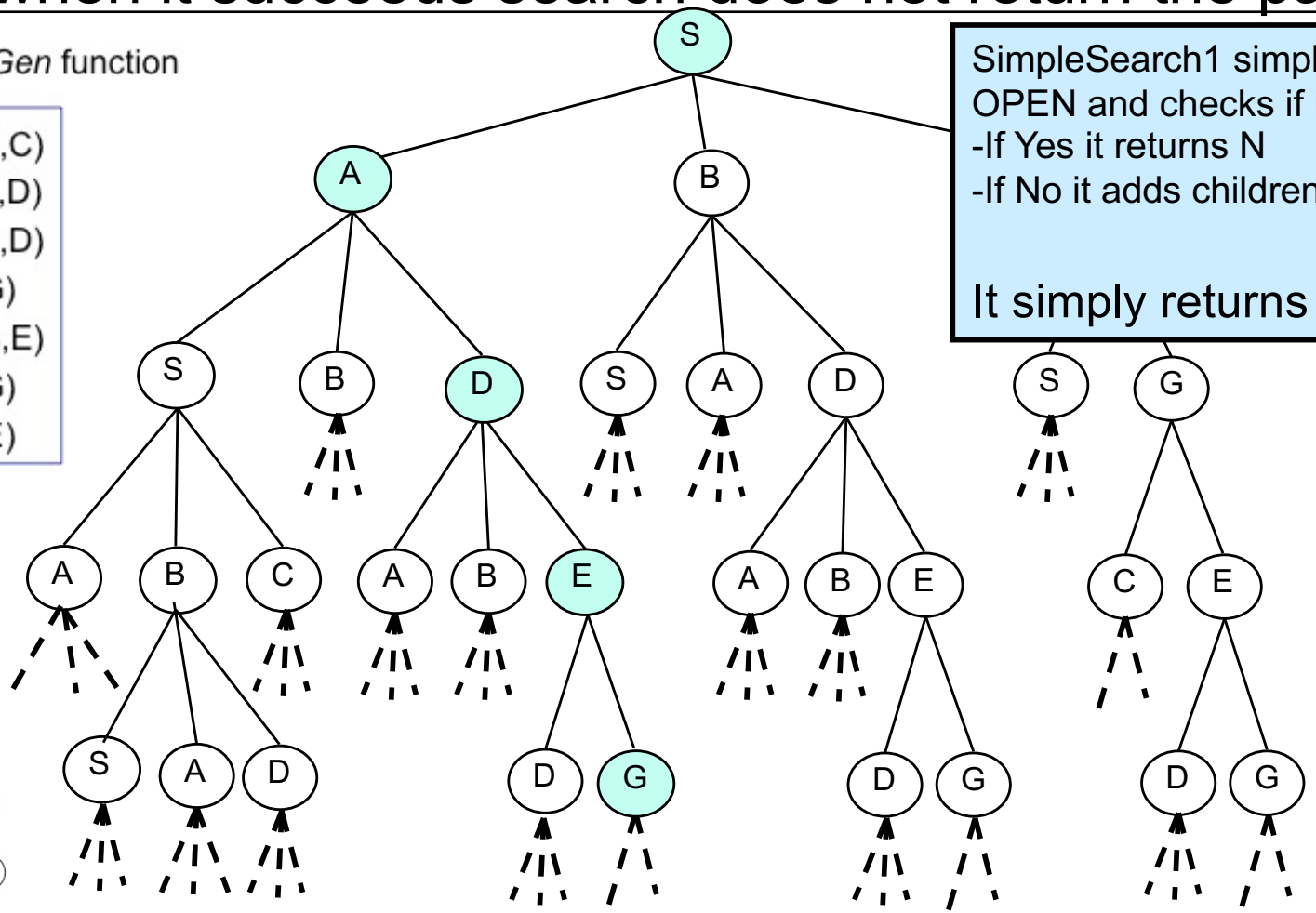



Each node is added exactly once to the search tree

# Even when it succeeds search does not return the path

The *MoveGen* function

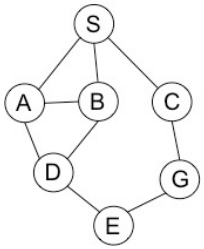
S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



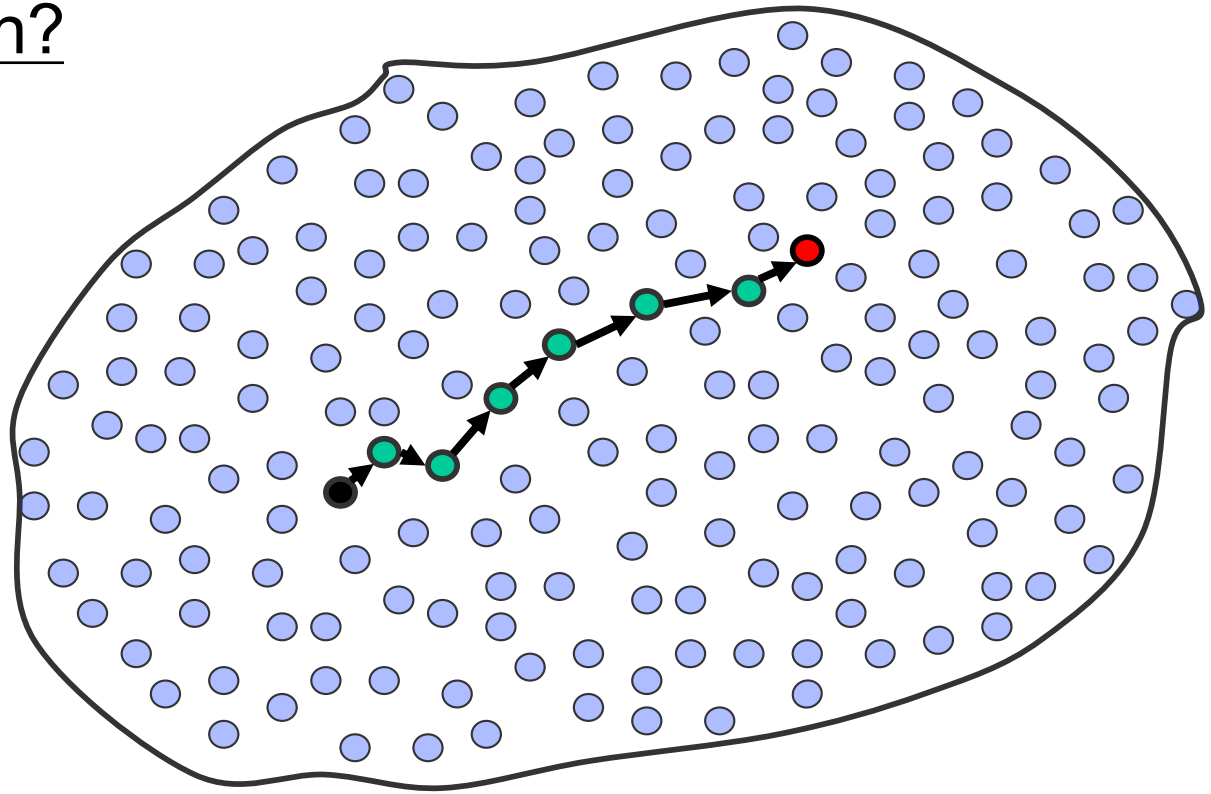

SimpleSearch1 simply picks a node N from OPEN and checks if it is the goal.
-If Yes it returns N
-If No it adds children of N to OPEN

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

## Planning problems

Goal is known or described, path is sought

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 sought

Some examples

N-queens

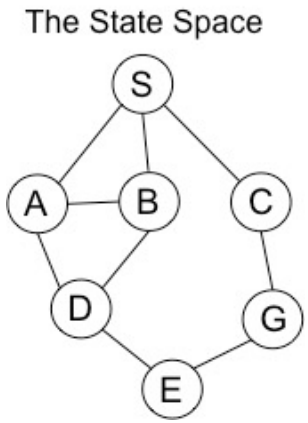
A crossword puzzle

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.

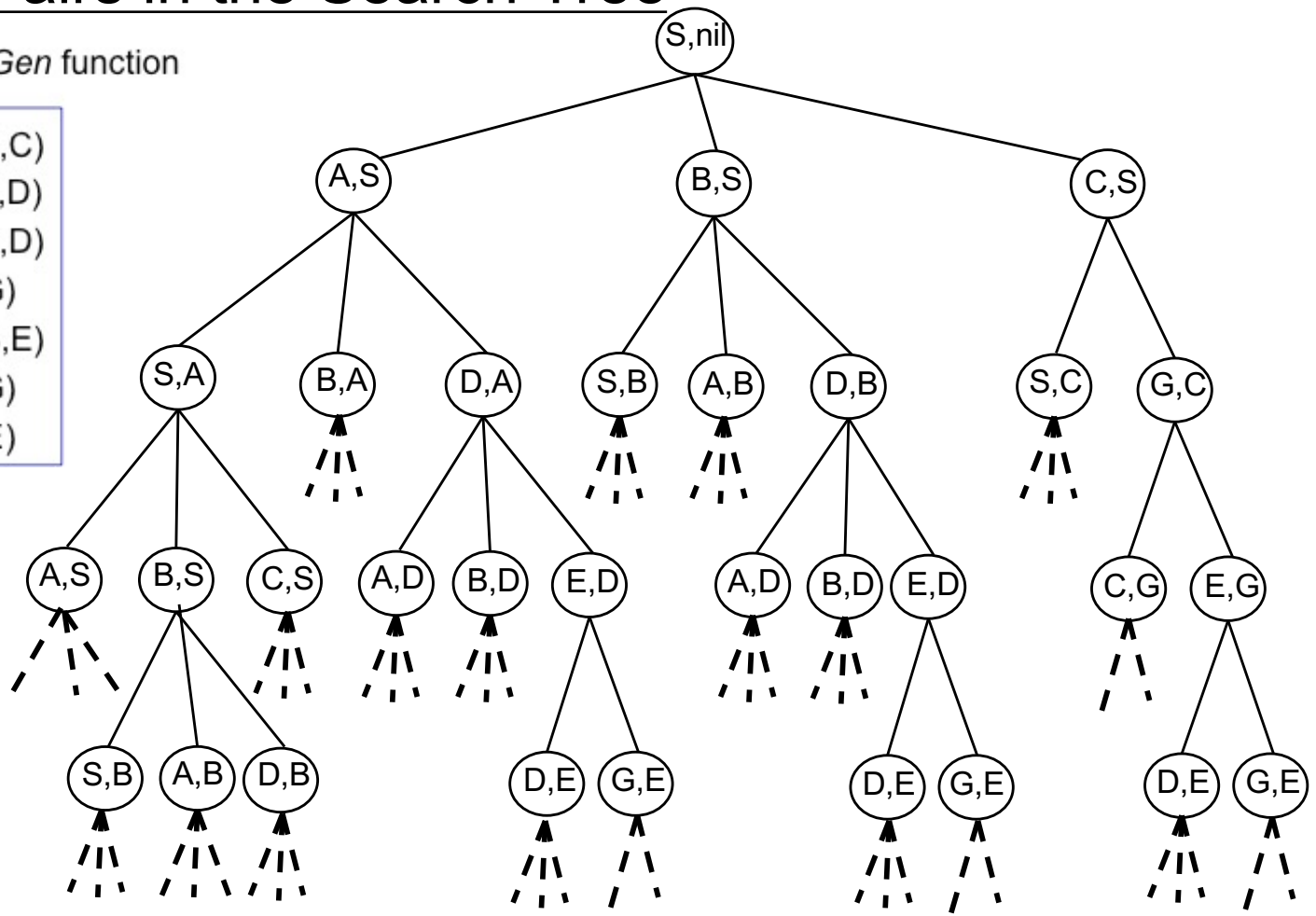
One way is to modify the search space node to store the entire path in the node itself.

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

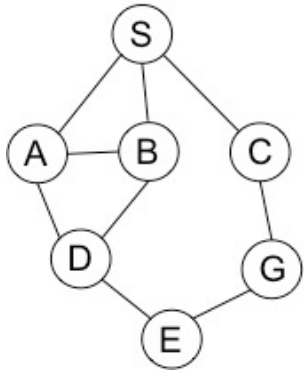
# NodePairs in the Search Tree

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)



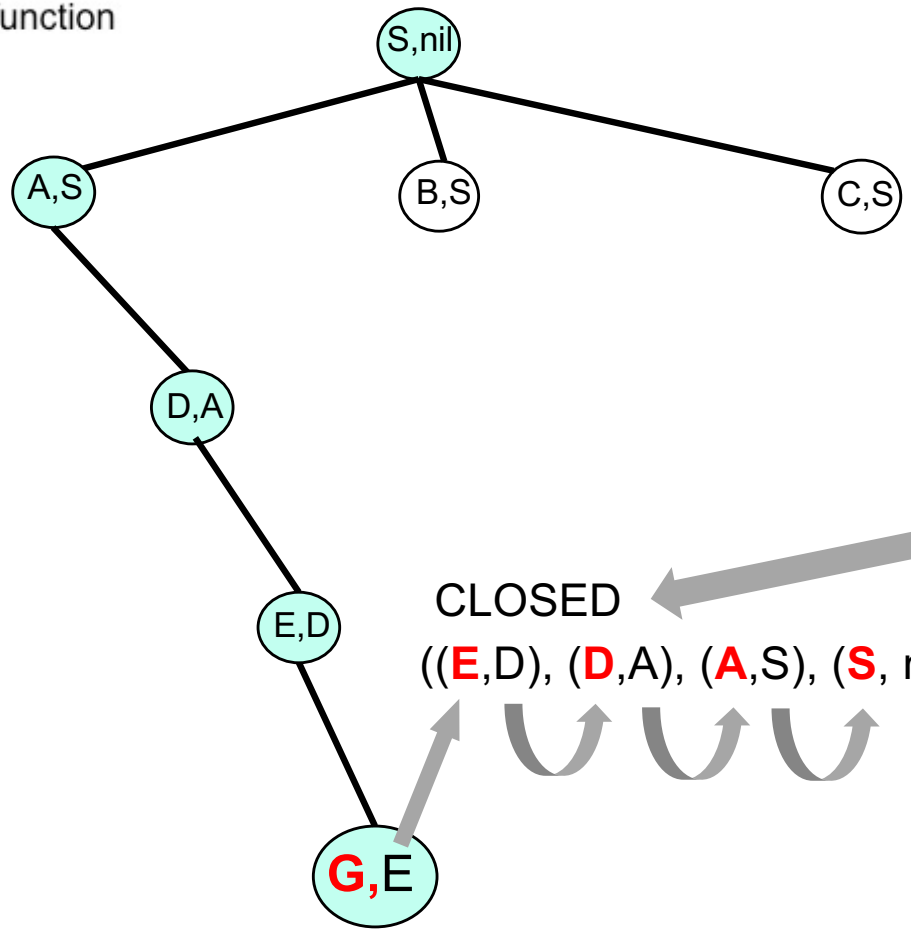

The State Space



# Reconstructing the Path

The *MoveGen* function

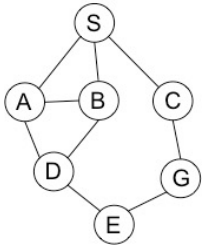
S → (A,B,C)
A → (S,B,D)
B → (S,A,D)
C → (S,G)
D → (A,B,E)
E → (D,G)
G → (C,E)



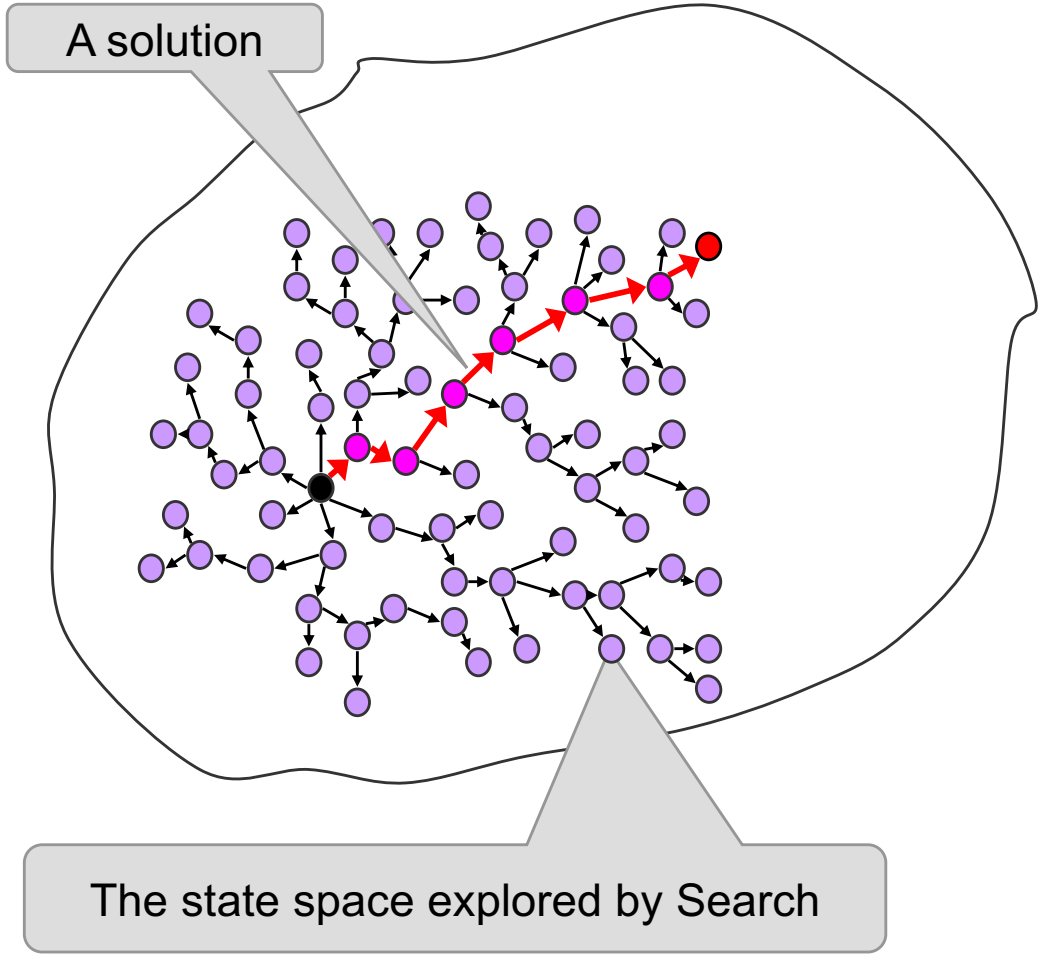

| OPEN | CLOSED |
|---|---|
| ((S,nil)) | ( ) |
| ((A,S),(B,S),(C,S)) | ((S,nil)) |
| ((D,A),(B,S),(C,S)) | ((A,S),(S,nil)) |
| ((E,D),(B,S),(C,S)) | ((D,A),(A,S), (S,nil)) |
| ((G,E),(B,S),(C,S)) | ((E,D),(D,A), (A,S), (S,nil)) |

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. $\text{LIST}_2 \leftarrow \text{ELEMENT}$ **:** $\text{LIST}_1$

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

# Some LIST notation (continued)

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

6. $[1] = 1 \mathbf{:} [\,]$

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

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

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

# Some LIST notation (continued)

10. $[3,2,1] \;=\; 3 \,\mathbf{:}\, [2,1] \;=\; 3 \,\mathbf{:}\, 2 \,\mathbf{:}\, [1] \;=\; 3 \,\mathbf{:}\, 2 \,\mathbf{:}\, 1 \,\mathbf{:}\, [\,]$

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

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

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

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

# 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. $\text{pair} \leftarrow (101, 102)$

3. $(a, b) \leftarrow \text{pair}$

4. $(a, \_) \leftarrow \text{pair}$

5. $a \leftarrow \textbf{first}\ \text{pair}$

# 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 and Breadth First Search

# 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 nodes 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

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

```
MAKEPAIRS(nodeList, parent)
1    if nodeList is empty
2        return empty list
3    else return (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

Traces back through Parent links and reconstructs the path found.

```
ReconstructPath (nodePair, CLOSED)
1.  (node, parent) ← nodePair
2.  path ←  node : [ ]
3.  while parent is not null
4.        path ← parent : path
5.        ( _, parent) ← FindLink(parent, CLOSED)
6.  return path
```

Finds that nodePair in CLOSED where node is the first element.

```
FindLink(node, CLOSED)
1.       if node = first head CLOSED
2.          return head CLOSED
3.       else return FindLink(node, tail CLOSED)
```

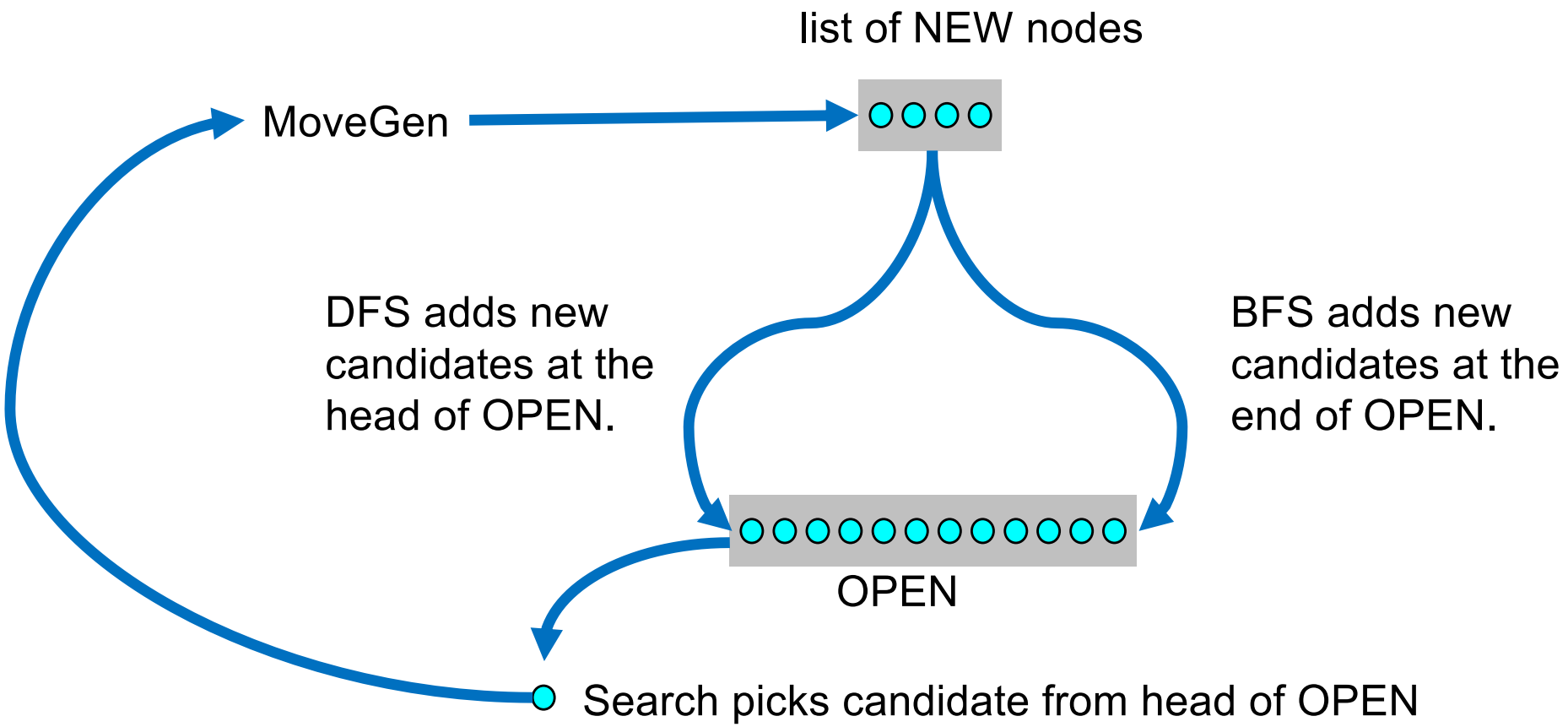
# Breadth First Search: OPEN = Queue

BFS(S)

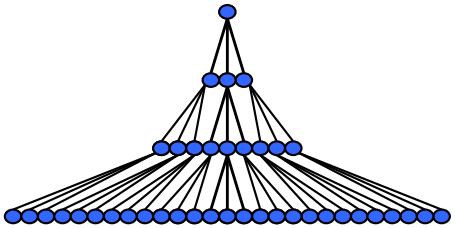
```
 1  OPEN ← (S, null) : [ ]
 2  CLOSED ← empty list
 3  while OPEN is not empty
 4        nodePair ← head OPEN
 5        (N, _) ← nodePair
 6        if GoalTest(N) = true
 7              return ReconstructPath(nodePair, CLOSED)
 8        else CLOSED ← nodePair : CLOSED
 9              children ← MoveGen(N)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

Only change

# Stack vs. Queue

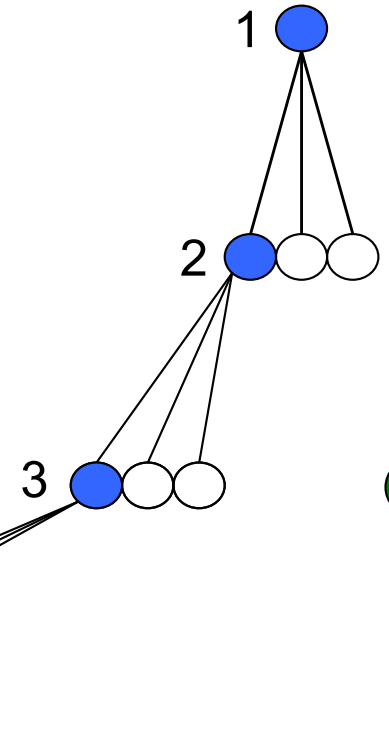


# DFS and BFS : Behavior



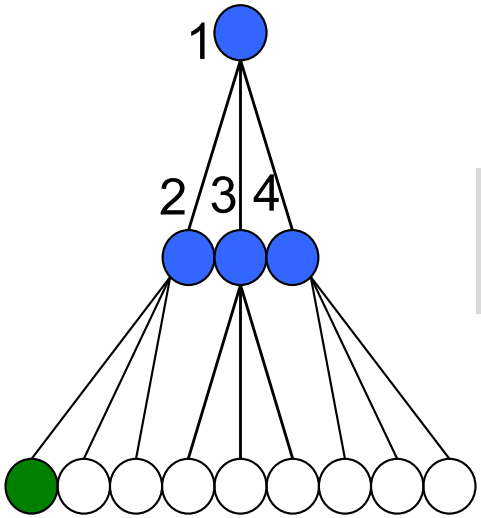
A Search Tree

DFS




Deepest Nodes First

BFS

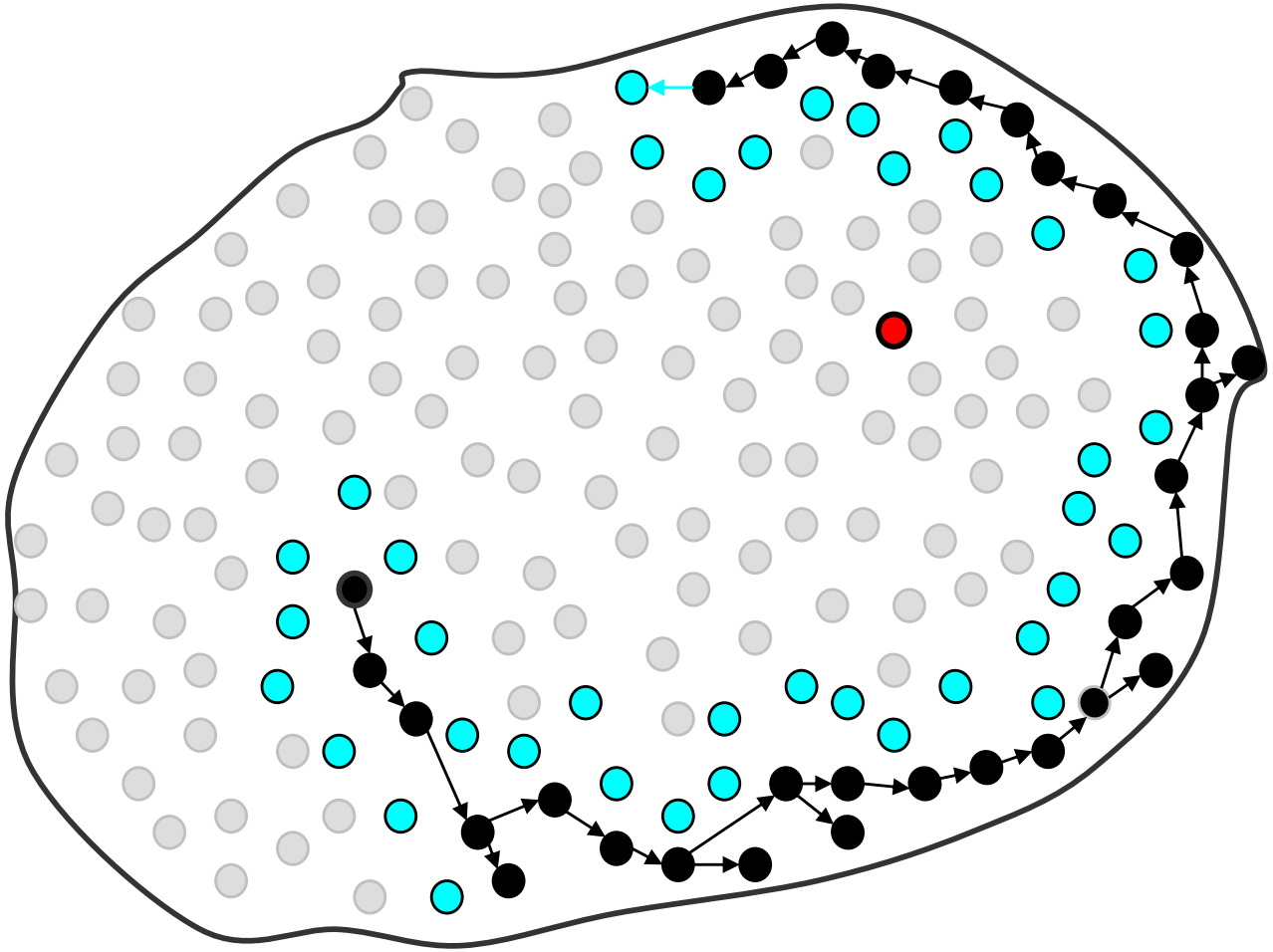



Shallowest Nodes First

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.

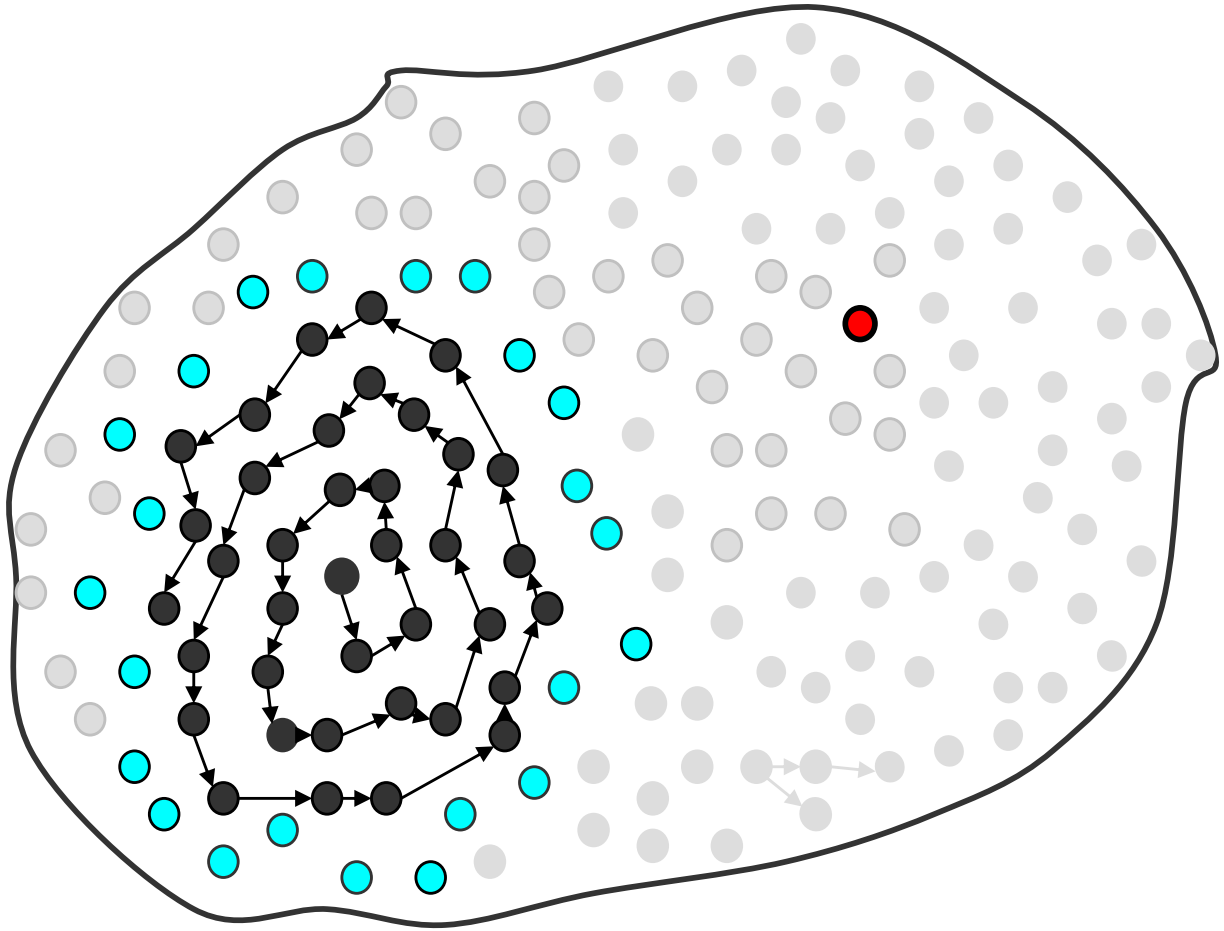
# Depth First Search

… dives down into the search tree

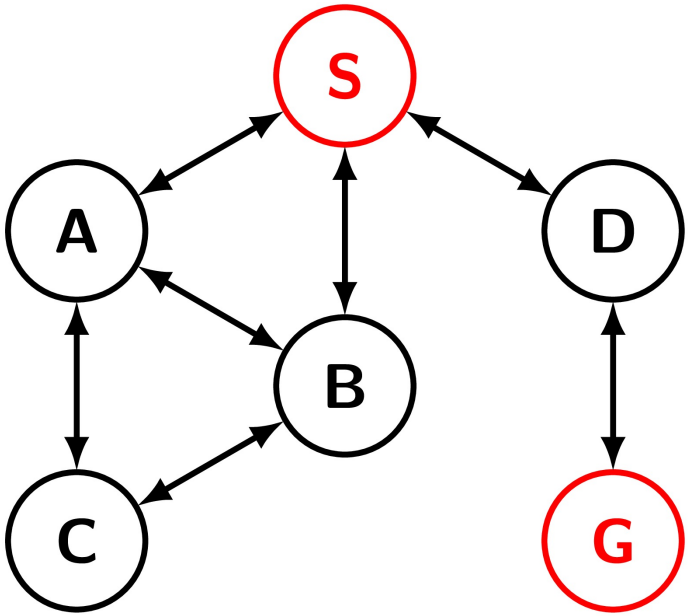


# Breadth First Search

… sticks close to the start node

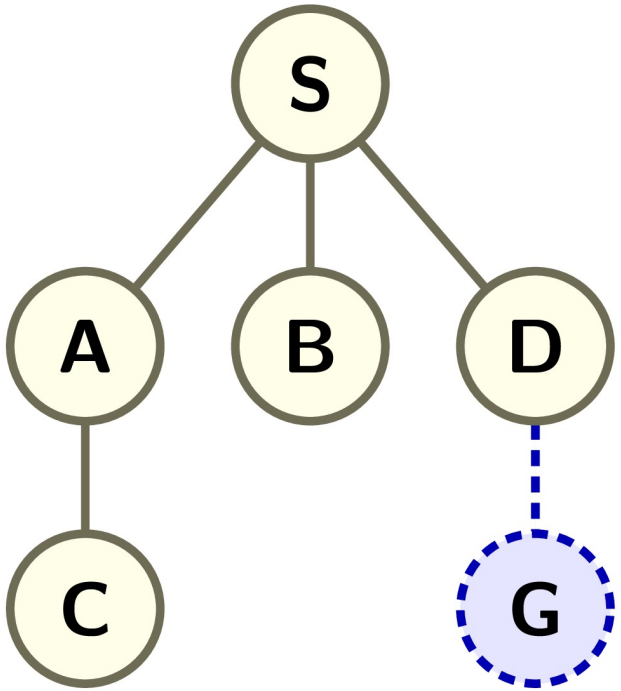


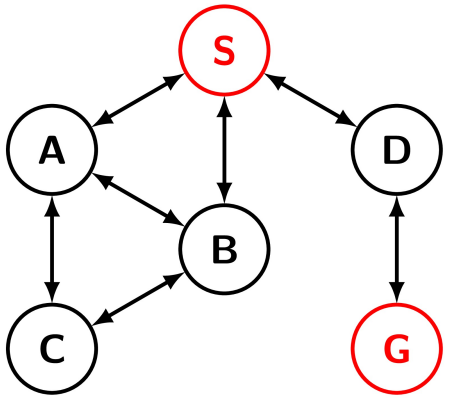
# A Tiny State Space




| 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: Prune nodes already in OPEN or CLOSED

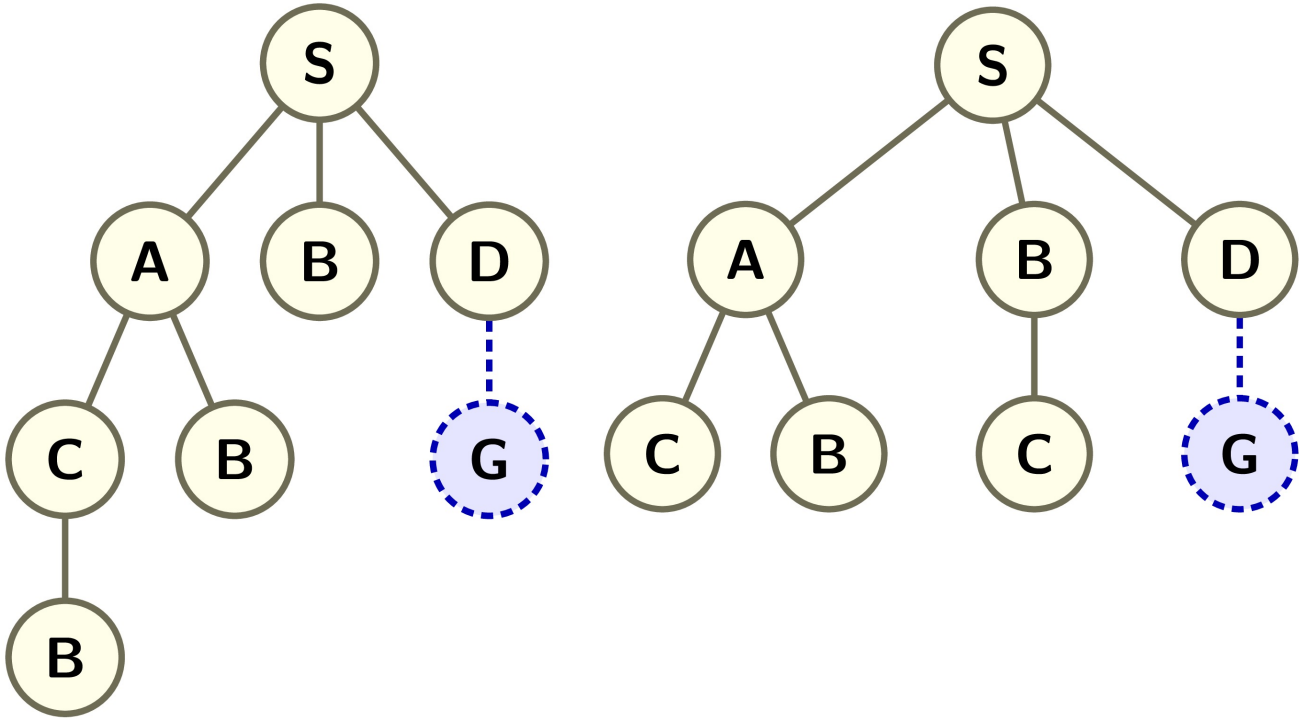



| X | | MoveGen(X) | GoalTest(X) |
|---|---|---|---|
| S | → | A,B,D | False |
| A | → | C,B,S | False |
| B | → | S,A,C | False |
| C | → | B,A | False |
| D | → | S,G | False |
| G | → | D | True |

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

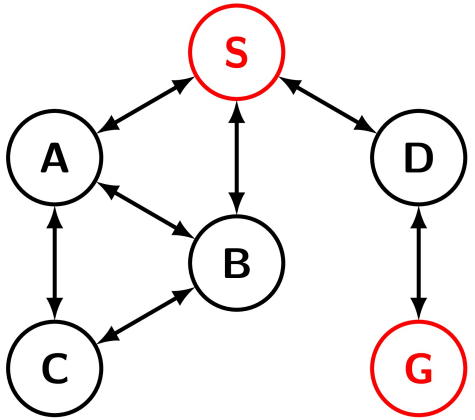
# Case 2: Only prune nodes already in CLOSED



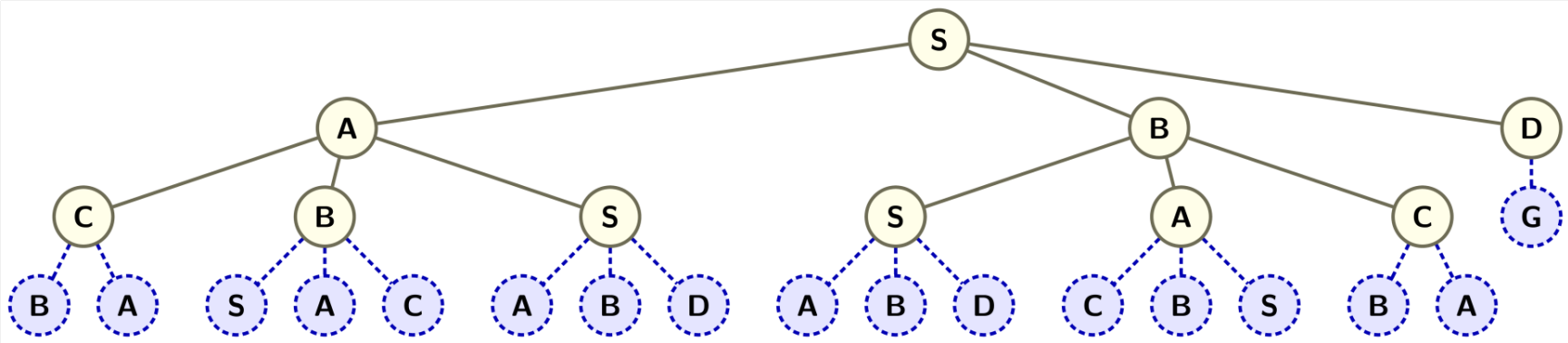

DFS search tree

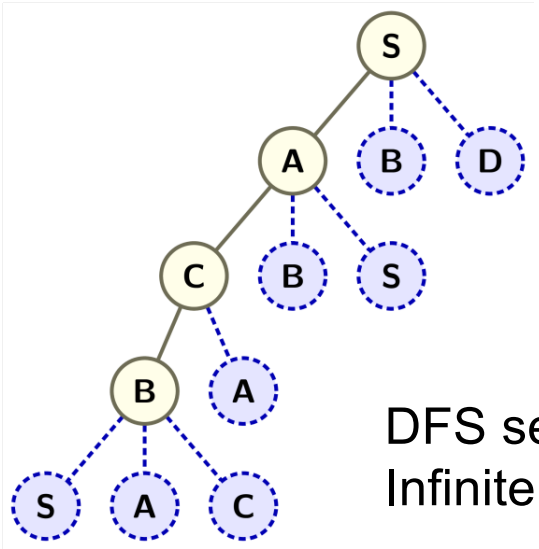



BFS search tree

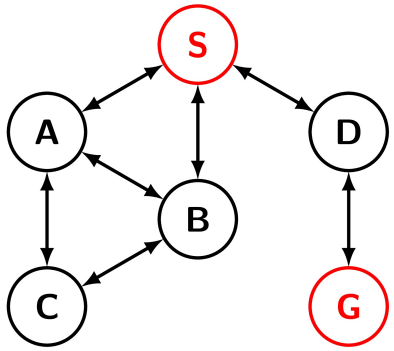


# Case 3: Do not prune any nodes from MoveGen



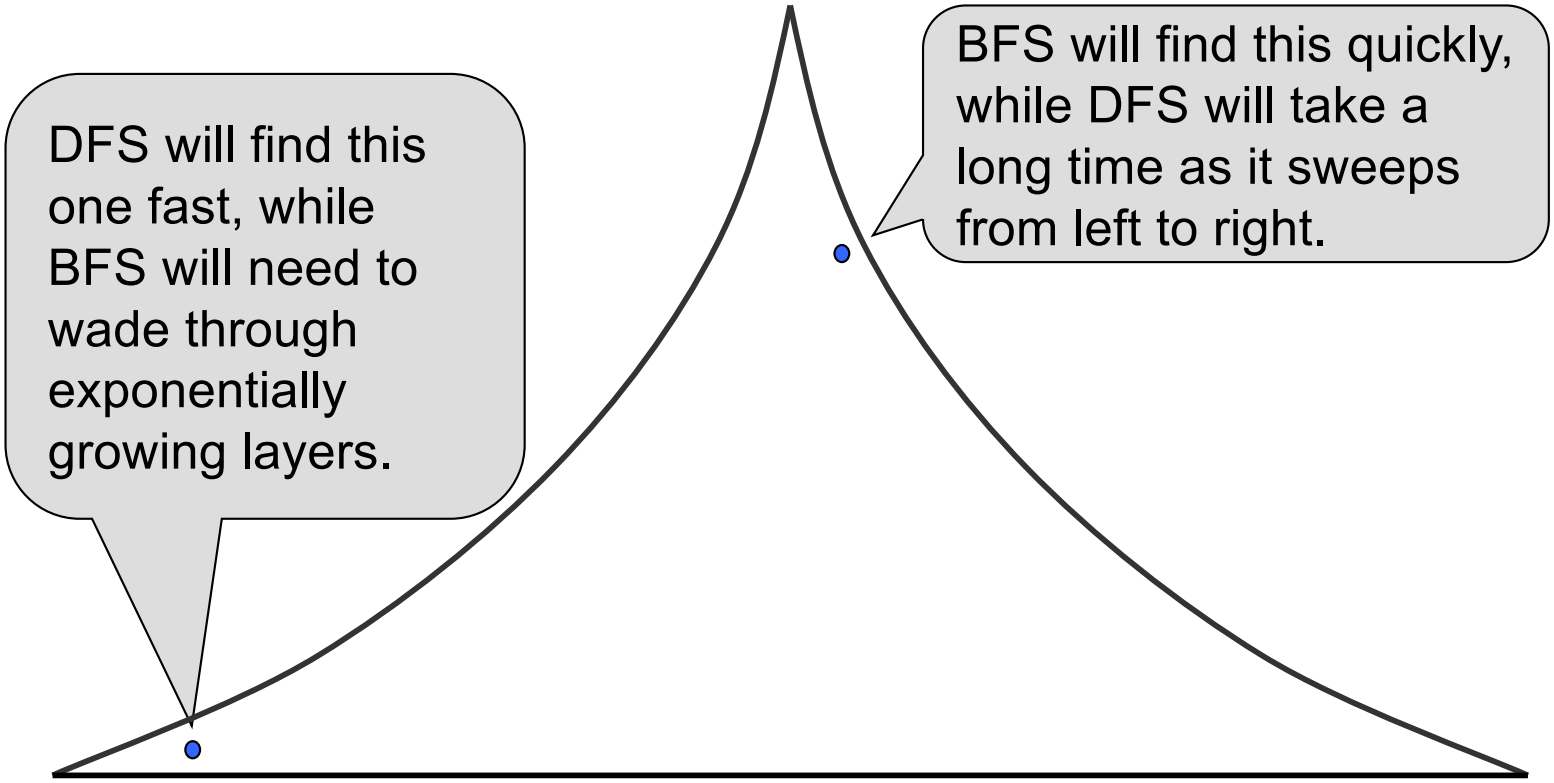

DFS search tree
Infinite loop



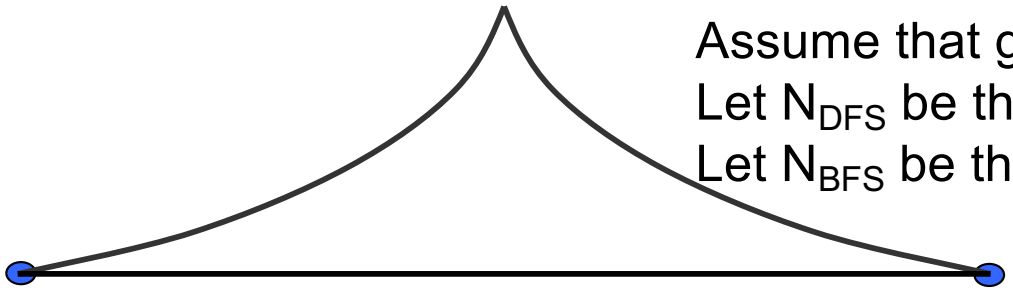

BFS search tree.
Terminates with shortest path

# Analysis

# 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

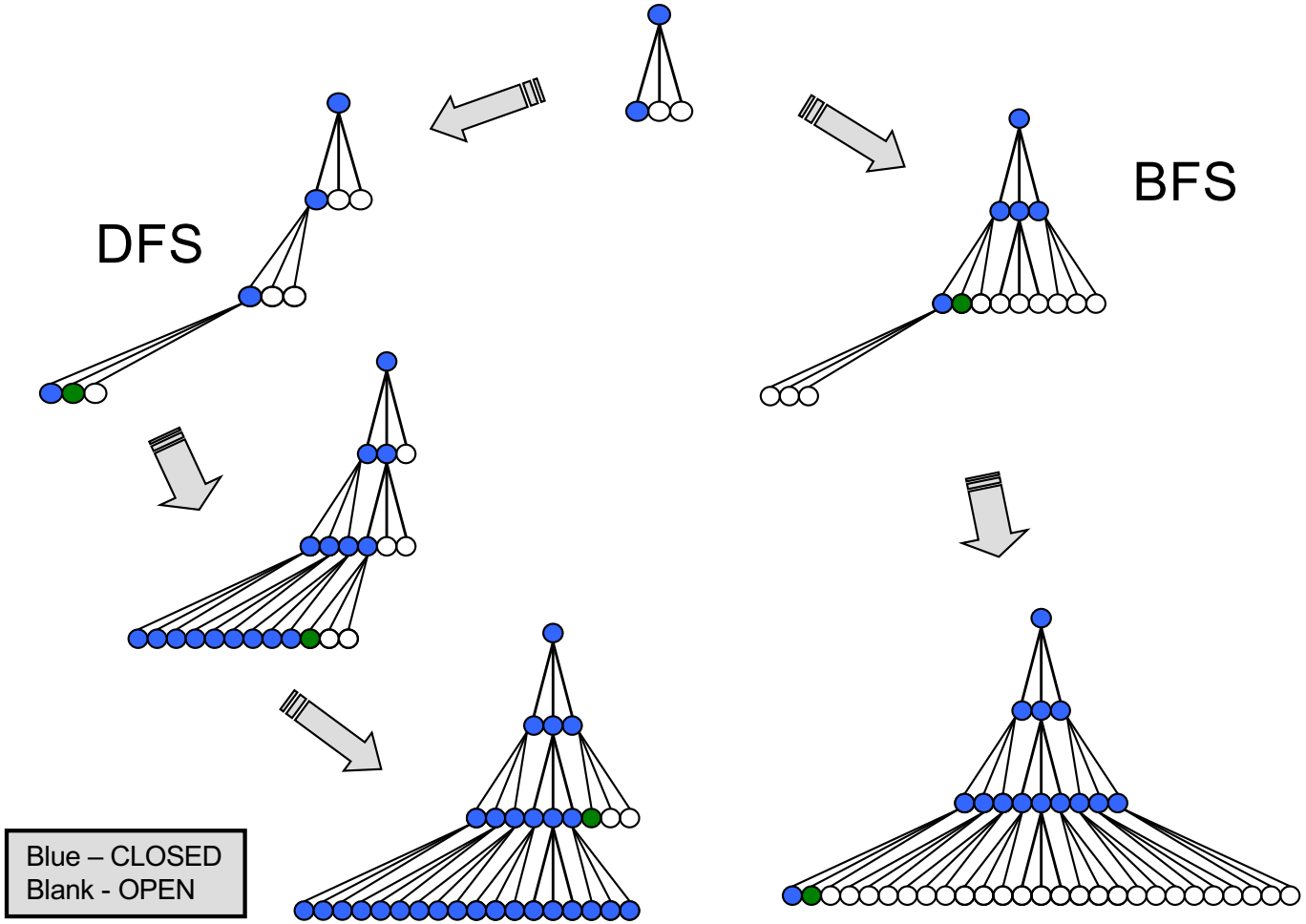
$$N_{DFS} = \frac{(d+1) + (b^{d+1}-1)/(b-1)}{2} \approx \frac{b^d}{2}$$

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

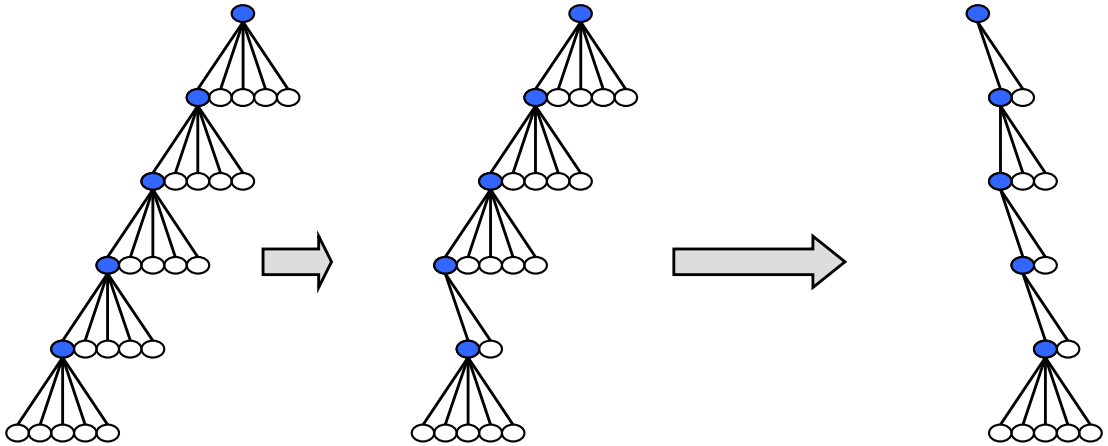
$$N_{BFS} \approx N_{DFS} \frac{(b+1)}{(b-1)}$$

Average time complexity is exponential

# DFS vs BFS : Space (Size of OPEN)



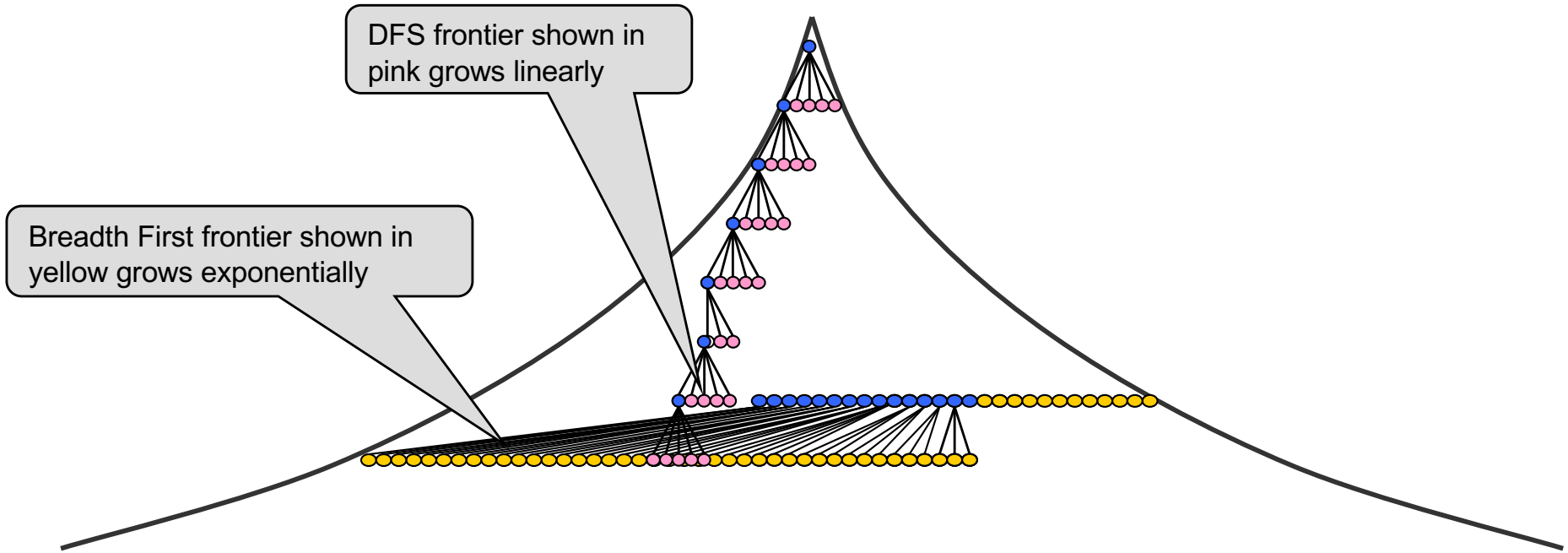
# DFS : The OPEN list



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.

# Search Frontiers




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

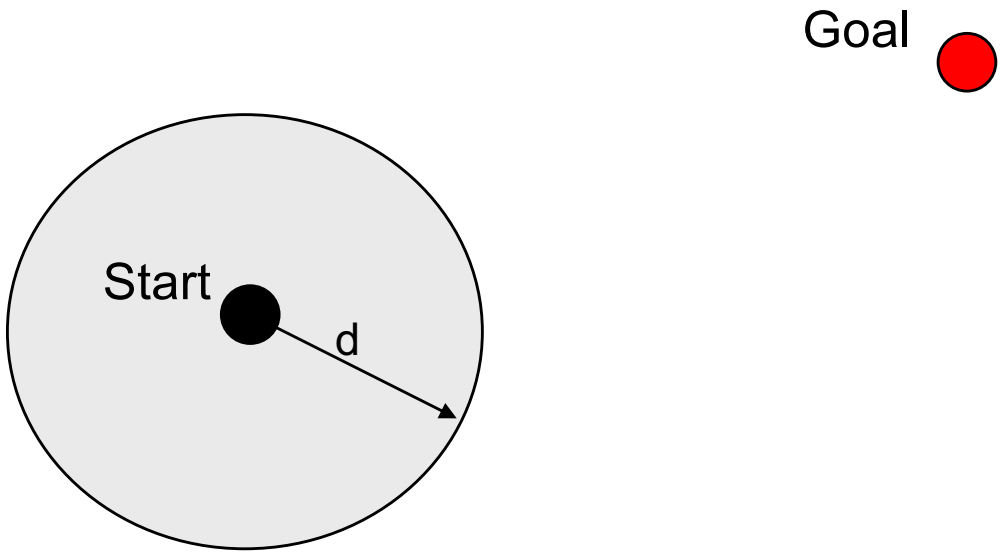
# Depth First vs. Breadth First

| | 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?

# The best of both

# 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
```

A triple, even if we still call it a nodePair

Search only within the depth bound

# 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)
```

Question: Why count nodes?

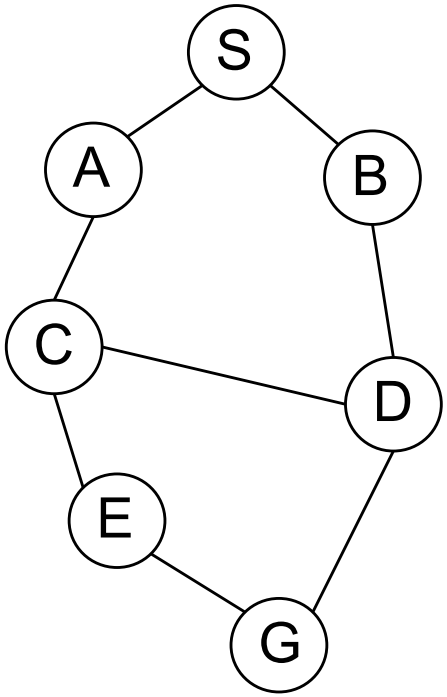
# Depth First Iterative Deepening (DFID)

```
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
```

Why?

# Does DFID return shortest path?

The State Space

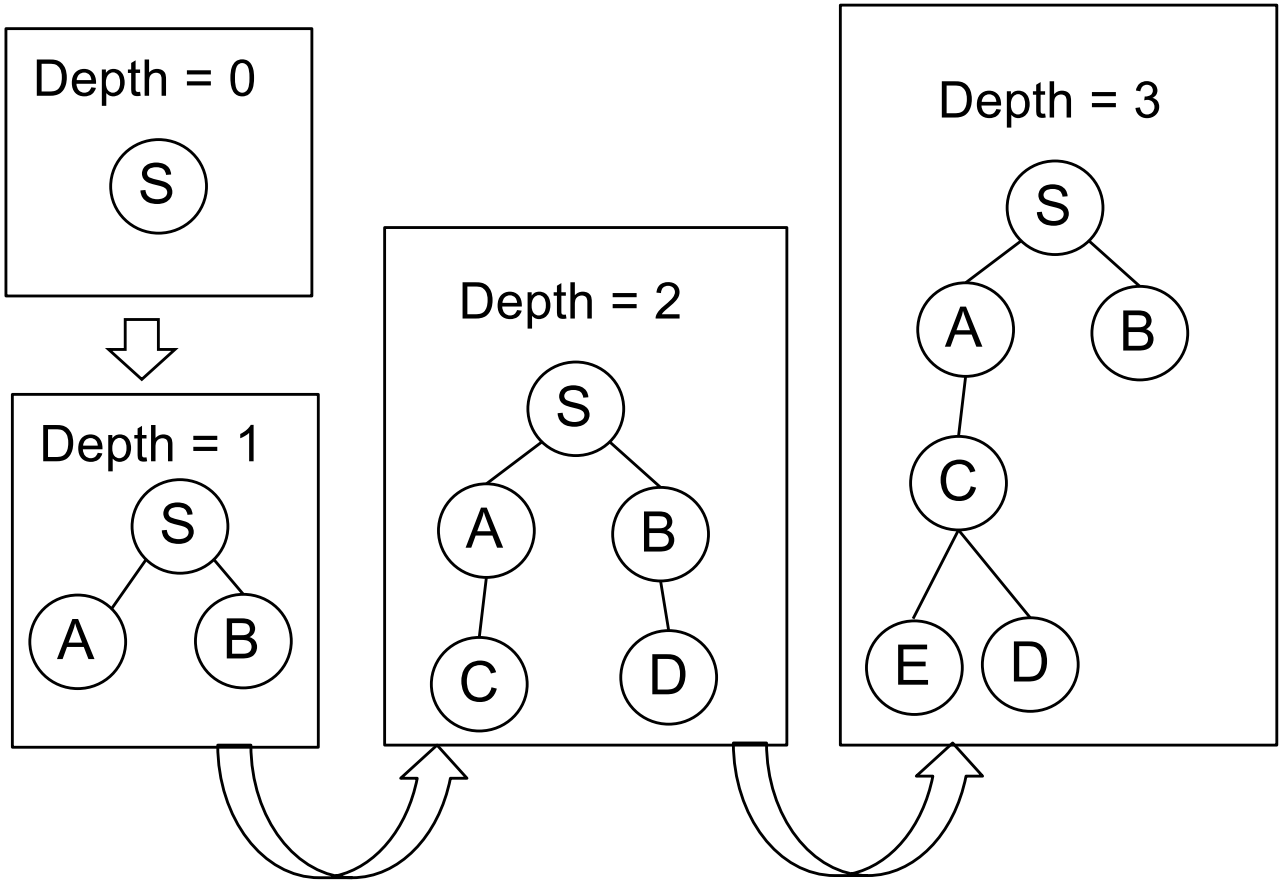



The *MoveGen* function

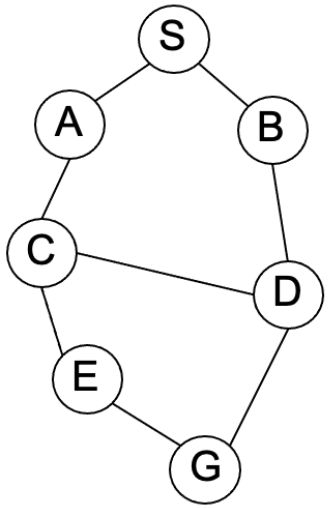
S → (A,B)
A → (S,C)
B → (S,D)
C → (E,D,A)
D → (B,C,G)
E → (C,G)
G → (D,E)

This was pointed out by Siddharth Sagar a student at IIT Dharwad in 2019

# Careful with CLOSED in DFID

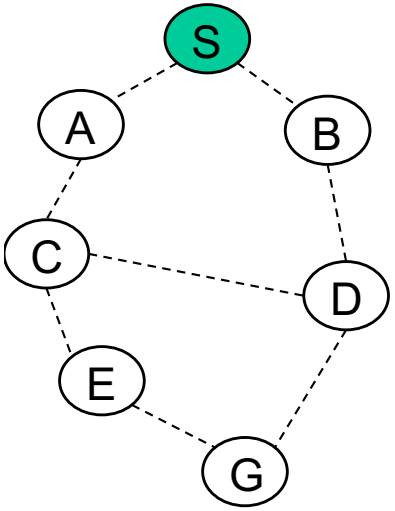



The State Space

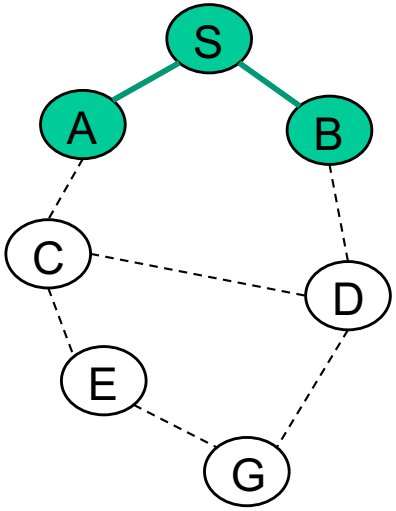



When depth = 3 node D is not generated as a child of B!

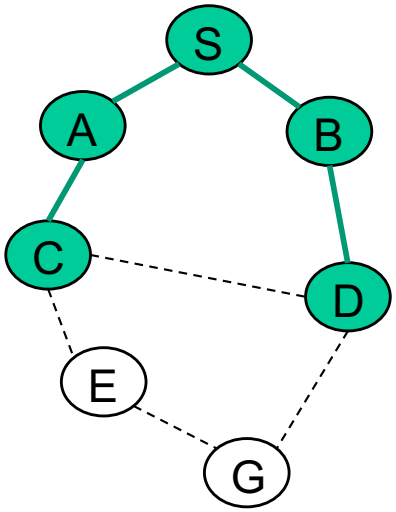
# DFID checking for loops with CLOSED



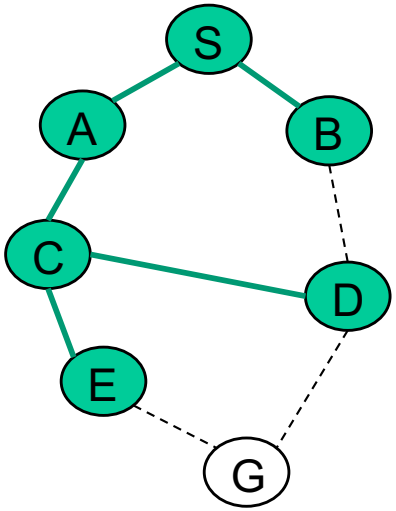

Paths of length 0



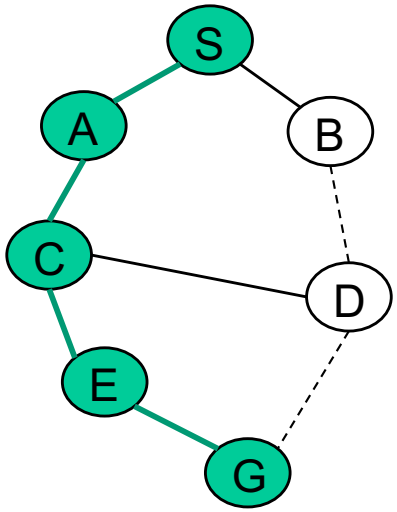

Paths of length 1




Paths of length 2




Paths of length 3




Paths of 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.

# Without CLOSED DFID finds the shortest path…

… but how does one reconstruct it?

Have we thrown the baby out with the bathwater?

Points to note –

- We still need the parent nodes to reconstruct the path
- We can still store *nodePairs* in CLOSED, and use them for reconstruction but have to be careful to choose the right *nodePair* as nodes can be reached through multiple parents
- Perhaps we can store the entire path at each node?
- For configuration problems the path is not needed. In fact the shortest "path" may have no importance except time complexity.
- But DFID enables *anytime* move generation in games like chess
  - DFID was invented by chess programmers

# 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

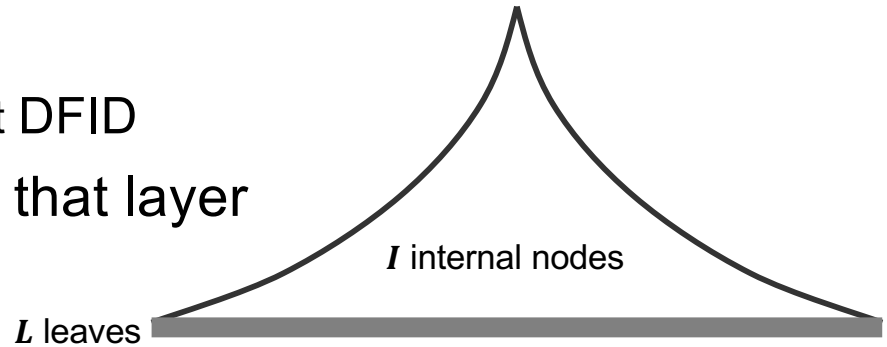
Is there a catch?

A series of depth bounded depth first searches *without checking* CLOSED
(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 Search 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 $b$

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

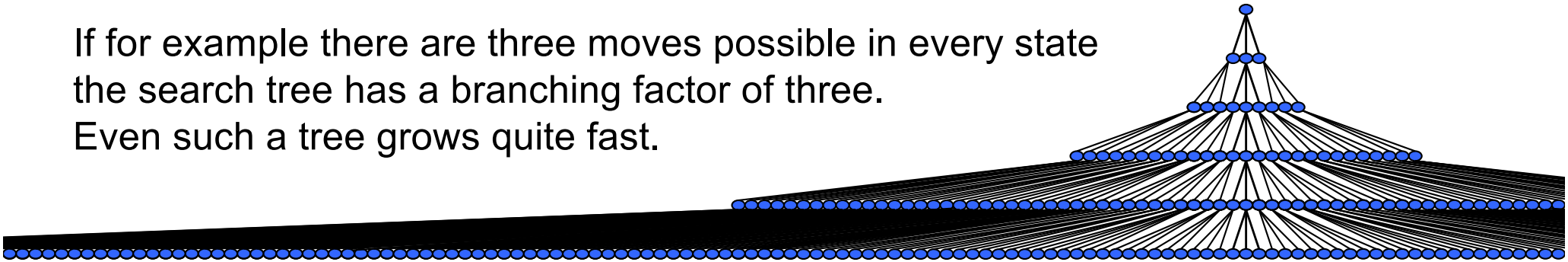
The extra cost is not significant!

# CombEx

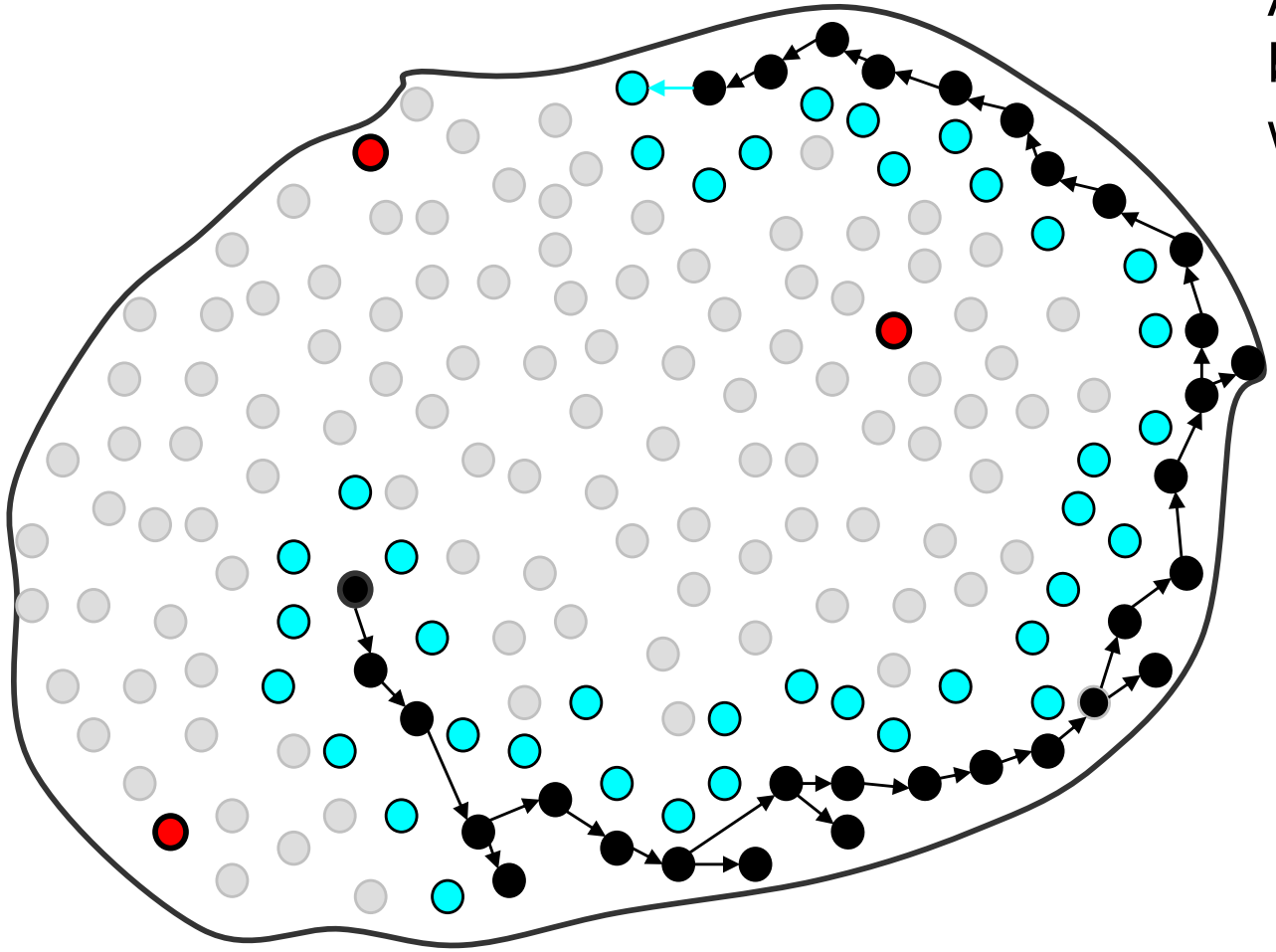
The Hydra like monster that AI fights is Combinatorial Explosion.

For every node that fails the goal test
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.

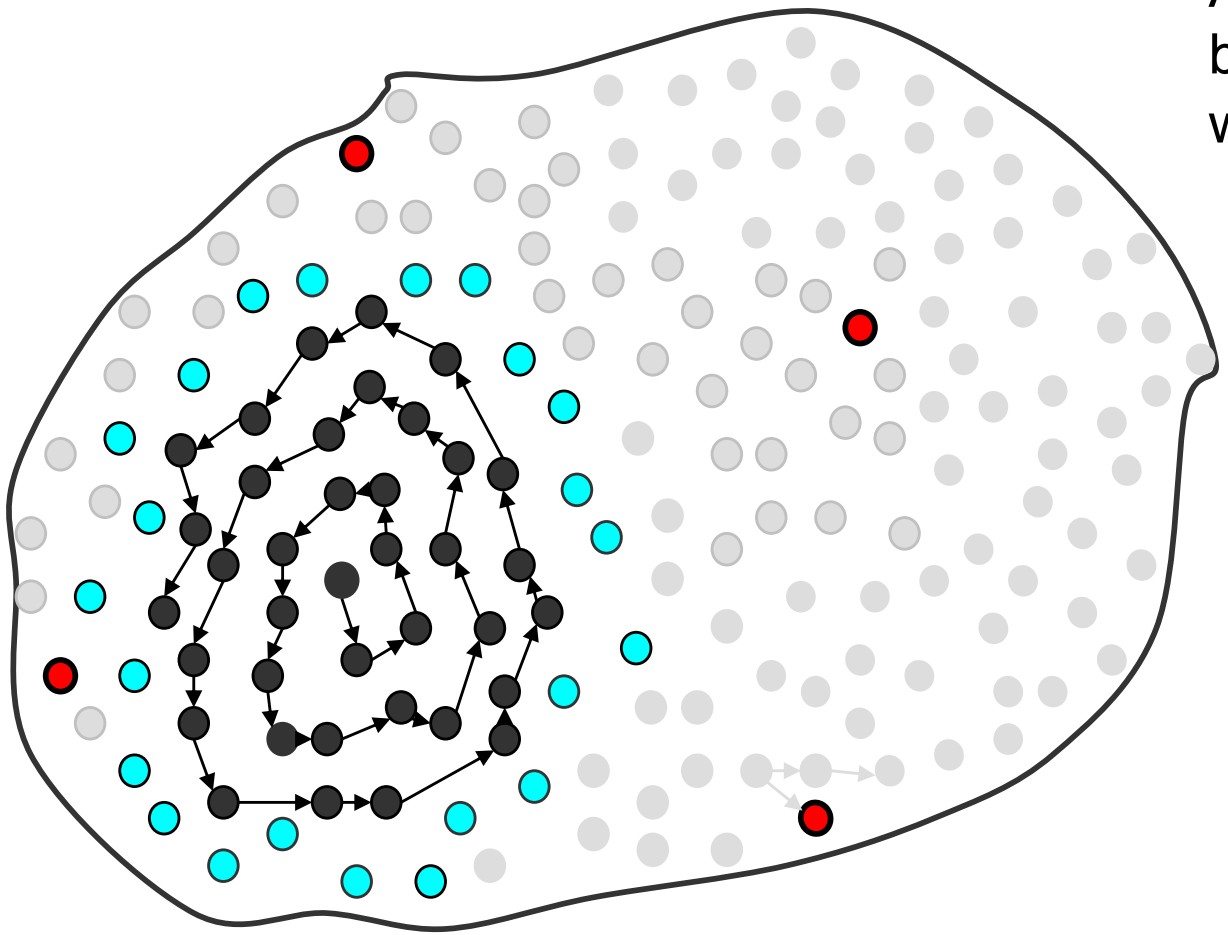


# Depth First Search



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….

All three algorithms we studied are blind or uninformed

# End
# Uninformed Search