# COSC367: Artificial Intelligence

This course introduces major concepts and algorithms in Artificial Intelligence. Topics include problem solving, reasoning, games, and machine learning.

Jordan Pyott

2021-05-06 16:57

# Contents

# Artificial Intelligence

## Course Information

The course covers core topics in AI including:

- uninformed and informed graph search algorithms,
- propositional logic and forward and backward chaining algorithms,
- declarative programming with Prolog,
- the min-max and alpha-beta pruning algorithms,
- Bayesian networks and probabilistic inference algorithms,
- classification learning algorithms,
- consistency algorithms,
- local search and heuristic algorithms such as simulated annealing, and population-based algorithms such as genetic search and swarm optimisation.

### Grades

Standard Computer science policy applies

- Average 50% over all assessment items
- Average at least 45% on all invigilated assessment items

Grading structure for course

- Assignments (5%)

    - Two Super Quiz's

- Quizzes (16.5%)

    - Weekly Quiz Assessments (1.5% ea)

- Lab Test (20%)
- Final Exam (58.5%)

## Textbooks / Resources

- Poole, David L.1958, Mackworth, Alan K; Artificial intelligence : foundations of computational agents; Cambridge University Press, 2010.
- Russell, Stuart J, Norvig, Peter; Artificial intelligence : a modern approach; 3rd ed; Prentice Hall, 2010.

**Readings**

**Lectures**

**Lecture One: Searching the State Space**

**What is state?**

- A state is a data structure that represents a possible configuration of the world *agent and environment*
- The **state space** is the set of all possible states for that problem
- actions change the state of the world

- Example: A vacuum cleaner agent in two adjacent rooms which can be either clean or dirty.

- Location = {left, right}
- Left-room-condition = {dirty, clean}
- Right-room-condition = {dirty, clean}
- State-space = Location
  × Left-room-condition
  × Right-room-condition

In this example, each state is represented by a triple (3-tuple).
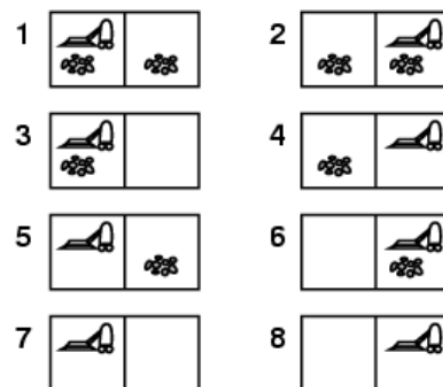
**Figure 1:** State space example one

State can also be represented as a graph *both directed and undirected*

- Example: Suppose the vacuum cleaner agent can take the following actions: L (go left), R (go right), S (suck).
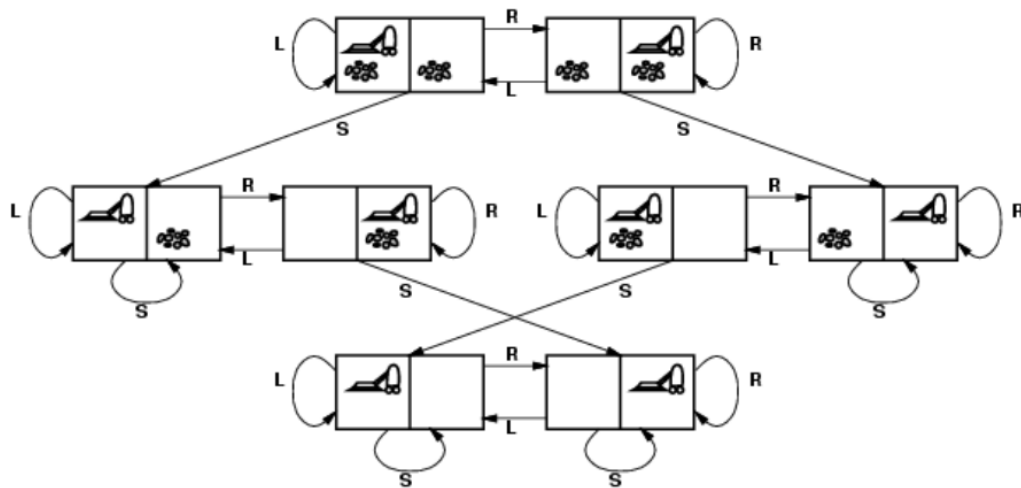


**Figure 2:** State space graph simplified

- Many problems in AI can be abstracted to the problem of finding a path in a directed graph
- Notation we use is **Nodes** and **arcs** for **vertices** and **edges** in a graph

**Explicit vs Implicit graphs**

- In **explicit graphs** nodes and arcs are readily available, they are read from the input and stored in a data structure such as an adjacency list/matrix.

  - the entire graph is in memory.
  - the complexity of algorithms are measured in the number of nodes and/or arcs.

- In **implicit graphs** a procedure `outgoing_arcs` is defined that given a node, returns a set of directed arcs that connect node to other nodes.

  - The graph is generated as needed *due to the complexity of the graphs.*
  - The complexity is measured in terms of the depth of the goal state node or *how far do we have to get into the graph to find a solution*.

**Explicit graphs in quizzes**

- In some exercises we use small explicit graphs to stydy the behaviour of various frontiers
- Nodes are specified in a set

- Edges are specified in a list

  - pairs of nodes, or triples of nodes (in a tuple)

**Searching graphs**

- We will use generic search algorithms: given a graph, start nodes, and goal nodes, incrementally explore paths from the start nodes.
- Maintain a **frontier** of paths that have been explored

  - frontier: paths that we have already explored

- As search proceeds, the frontier is updated and the graph is explored until a goal node is found.
- The order in which paths are removed and added to the frontier defines the search strategy
- A **search tree** is a tree drawn out of all the possible actions in terms of a tree.

  - How do we handle loops? *Covered in next lecture*
  - In the search tree outlined below, you can see that the *end of paths on frontier* represents a BFS relationship note this is not always the case.
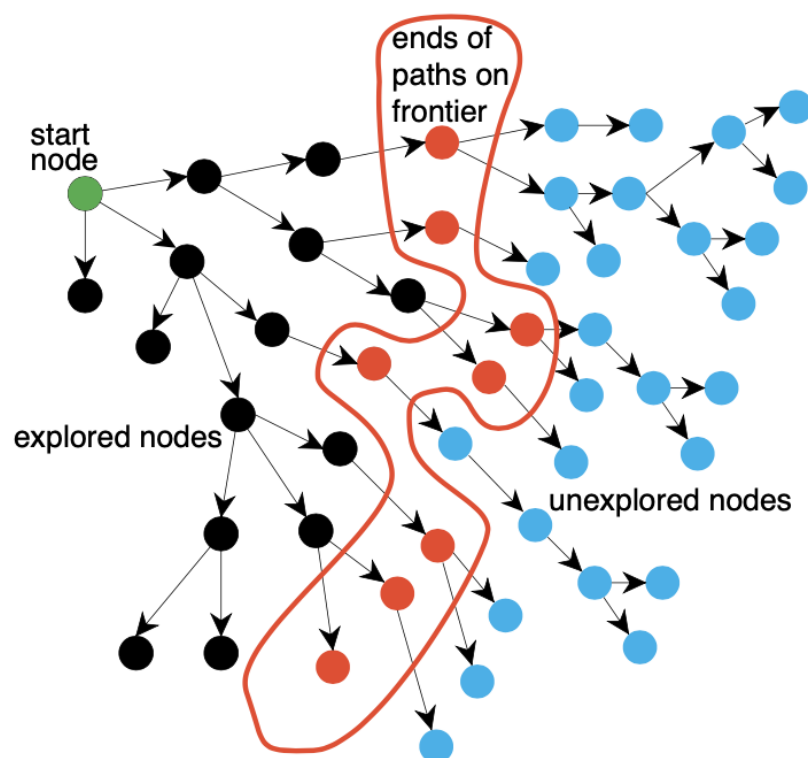


**Figure 3:** search tree

**Generic graph search algorithm**

**Input:** a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if $n$ is a goal node
$frontier := \{\langle s \rangle : s \text{ is a start node}\}$;
**while** $frontier$ is not empty:
    **select** and **remove** path $\langle n_0, \ldots, n_k \rangle$ from $frontier$;
    **if** $goal(n_k)$
        **return** $\langle n_0, \ldots, n_k \rangle$;
    **for every** neighbor $n$ of $n_k$
        **add** $\langle n_0, \ldots, n_k, n \rangle$ to $frontier$;
**end while**

**Figure 4:** Generic Search

> NOTE: you will have to use what ever data structure for the seach you are using (BFS use a queue), (DFS use a stack).

In the generic algorithm, neighbours are going to use the method `outgoing_arcs`, we are given this algorithm in the form of a python module.

**Depth-first search**

- In order to perform DFS, the generic graph search must be used with a stack frontier *LIFO*
- If the stack is a python list, where each element is a path, and has the form […, p, q]

  - *q* is selected and popped
  - of the algorithm continues then paths that extend *q* are pushed (appended) to the stack
  - *p* is only selected when all paths from *q* have been explored.

- As a result, at each stage the algorithm expands the deepest path
- The orange nodes in the graph below are considered the frontier nodes
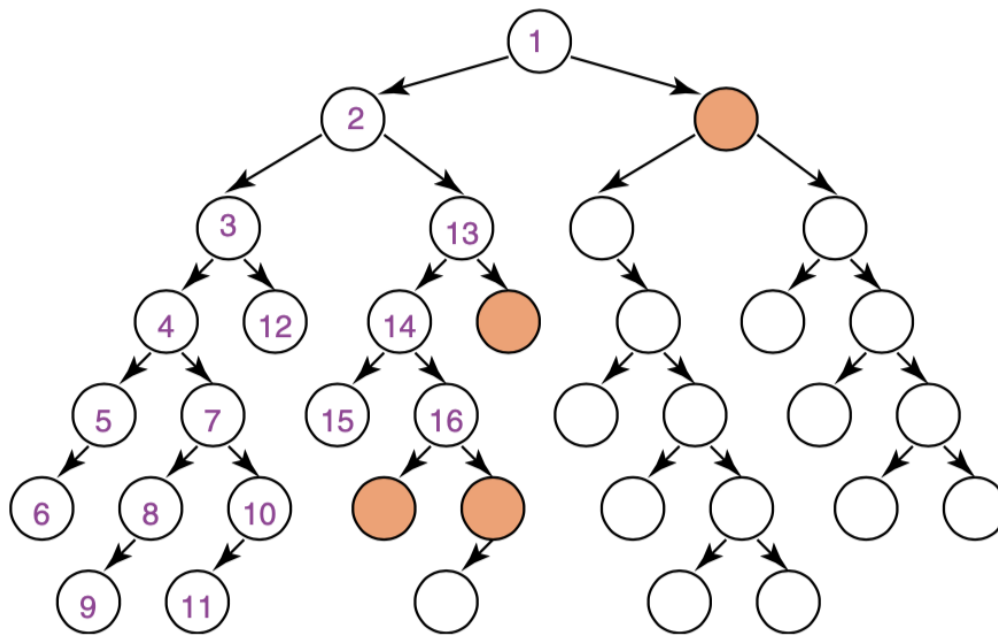
**Figure 5:** DFS

- DFS does not guarantee a solution without pruning, due to the fact that we can have infinite loops
- It is not guaranteed to complete if it does not use pruning

**A note on complexity**

Assume a finite search tree of depth *d* and branching factor of *b*:

- What is the time complexity?
    - It will be exponential: $O(b^d)$
- What is the space complexity?
    - It will be linear: $O(bd)$

**How do we trace the frontier**

- starting with an empty frontier we record all the calls to the frontier: to add or get a path we dedicate one line per call
- When we ask the frontier to add a path, we start the line with a + followed by the path that has been added
- When we ask for a path from the frontier we start the line with a − followed by the path being removed

- When using a priority queue, the path is followed by a comma and then the key *e.g, cost, heuristic, f-value, …*
- The lines of the trace should match the following regular expression `^[+-][a-z]+(,\d+)?!?$`
- We stop when we **remove** a path from the trace

Given the following graph
    nodes={a, b, c, d},
    edge_list=[(a,b), (a,d), (a, c), (c, d)],
    starting_nodes = [a],
    goal_nodes = {d}

trace the frontier in depth-first search (DFS).

Answer:
```
+ a
- a
+ ab
+ ad
+ ac
- ac
+ acd
- acd
```

**Figure 6:** DFS trace using generic algorithm

**Breath-first search**

- In order to perform BFS, the generic graph search must be used with a queue frontier *FIFO*.
- If the queue is a python deque of the form [p,q,…,r], then

    - p is selected (dequeued)
    - if the algorithm continues then paths that extend *p* are enqueued *appended* to the queue after *r*

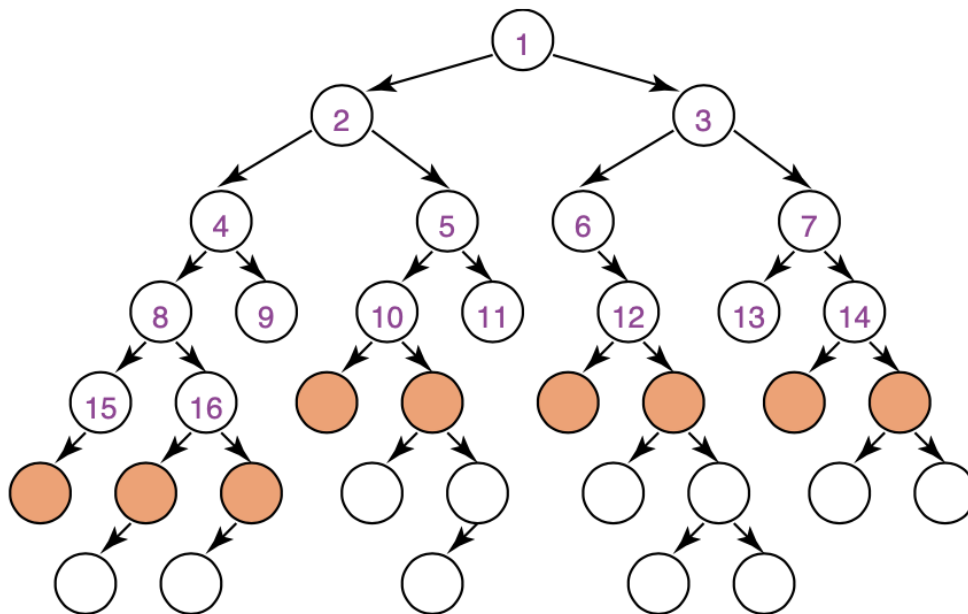- As a result, at each state the algorithm expands the shallowest path.

**Figure 7:** BFS Illustration of search tree

- BFS **does** guarantee to find a solution with the fewest arcs if there is a solution
- It will complete
- It will not halt due to some graphs having *cycles, with no pruning*

**A note on complexity**

> BFS has higher complexity than DFS

- What is the time complexity?

    – It will be exponential: $O(b^d)$

- What is the space complexity?

    – It will be linear: $O(b^d)$

Given the following graph

```
nodes={a, b, c, d},
edge_list=[(a,b), (a,d), (a, c), (c, d)],
starting_nodes = [a],
goal_nodes = {d}
```

trace the frontier in breadth-first search (BFS).

Answer:

```
+ a
- a
+ ab
+ ad
+ ac
- ab
- ad
```

**Figure 8:** BFS trace using generic algorithm

**Lowest-cost-first search**

- The cost of a path is the sum of the costs of its arcs
- This algorithm is very similar to Dijkstra's except modified for larger graphs
- LCFS selects a path on the frontier with the lowest cost
- The frontier is a priority queue ordered by path cost

  - A priority queue is a container in which each element has a priority *cost*
  - An element with a higher priority is always selected/removed before an element with a lower priority
  - In python we can use the heapq you will need to store objects in a way that these properties hold

- LCFS finds an optimal solution: a least-cost path to a goal node.
- Another name for this algorithm is *uniform-cost search*.

NOTE: For an example of this queue, see Lecture One: 1:45 time stamp

Given the following graph
    nodes={a, b, c, d, g},
    edge_lists=[(a,b,4), (a,c,2), (a,d,1),
                (b,g,4), (c,g,2), (d,g,4)],
    starting_nodes = [a],
    goal_nodes = {g}

trace the frontier in lowest-cost-first
search (LCFS).

Answer:
```
+ a,   0
- a,   0
+ ab,  4
+ ac,  2
+ ad,  1
- ad,  1
+ adg, 5
- ac,  2
+ acg, 4
- ab,  4
+ abg, 8
- acg, 4
```

**Figure 9:** LCFS trace generic

**Lecture Two: Searching the State Space (part two)**

**Pruning**

- This is our method to deal with cycles and multiple paths.
- this means we can have wasted computation and cycles in our graph

Principle: Do not expand paths to nodes that have already been expanded

**Pruning Implementation**

- The frontier keeps track of expanded or *closed* nodes
- When adding a new path to the frontier, it is only added if another path to the same end-node has not already been expanded, otherwise the new path is discarded (*pruned*)
- When asking for the **next path** to be returned by the frontier, a path is selected and removed but it is returned only if the end-node has not been expanded before, otherwise the path is discarded (pruned) and not returned. The selection and removal is repeated until a path is returned (or the frontier becomes empty). If a path is returned, its end-node will be remembered as an expanded node.

In frontier traces every time a path is pruned, we add an explanation mark ! at the end of the line

# Example: LCFS with pruning

Trace LCFS with pruning on the following graph:

```
nodes = {S, A, B, G},
edge_list=[(S,A,3), (S,B,1), (B,A,1), (A,B,1), (A,G,5)],
starting_nodes = [S],
goal_nodes = {G}.
```

Answer:
```
                    # expanded={}
+ S,0
- S,0        # expanded={S}
+ SA,3
+ SB,1
- SB,1       # expanded={S,B}
+ SBA,2
- SBA,2      # expanded={S,B,A}
+ SBAB,3!    # not added!
+ SBAG,7
- SA,3!      # not returned!
- SBAG,7     # expanded={S,B,A,G}
```

4

**Figure 10:** Example: LCFS with pruning

**How does LCFS behave?**

- LCFS explores increasing cost contours

  - Finds an optimal solution always
  - Explores options in every direction
  - No information about goal location

We are going to use a search heuristic, function h() is an estimate of the cost for the shortest path from node *n* to a goal node.

- *h* needs to be efficient to compute
- *h* can be extended to paths: $h(< n_0, ..., n_k) = h(n_k)$
- *h* is said to be admissible if and only if:

– $\forall n\, h(n) \geq 0$, $h$ is non-negative and $h(n) \leq C$ where C is the optimal cost of getting from $n$ to a goal node

NOTE: We will have to come up with our own heuristic for the assignment as it depends on context.
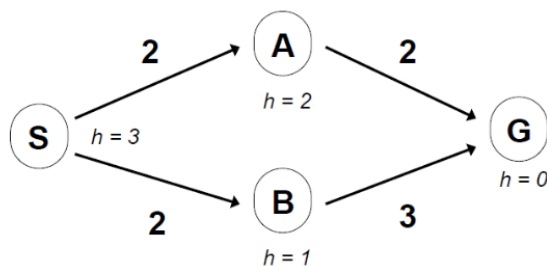
**Best-first Search**

- Idea: select the path whose end is closest to a goal node according to the heuristic function.
- Best-first search is a greedy strategy that selects a path on the frontier with minimal $h$-value
- Main drawback: this does not guarentee finding an optimal solution.

# Example: tracing best-first search

- Trace the frontier when using the best-first (greedy) search strategy for the following graph.

- The starting node is S and the goal node is G.

- Heuristic values are given next to each node.

- SA comes before SB.

**heuristic function**
h(S) = 3
h(A) = 2
h(B) = 1
h(G) = 0

Answer:

+ S,3

– S,3

+ SA,2

+ SB,1

– SB,1

+ SBG,0

– SBG,0

11

**Figure 11:** Tracing best-first search

**$A$ search strategy**

Properties:

- Always finds an optimal solution as long as:

- there is a solution
- there is no pruning
- the heuristic function is admissible

- Does it halt on every graph?

Idea:

- Don't be as wasteful as LCFS
- Don't be as greedy as best-first search
- Estimate the cost of paths as if they could be extended to reach a goal in the best possible way.

Evaluation function: $f(p) = cost(p) + h(n)$

- $p$ is a path, $n$ is the last node on $p$
- $cost(p)$ = cost of path $p$ *this is the actual cost from the starting node to node n*
- $h(n)$ = an estimate of the cost from $n$ to goal node
- $f(p)$ = estimated total cost of path through $p$ to goal node

The frontier is a priority queue ordered by $f(p)$

# Example: tracing A* search

- Trace the frontier when using the A* search strategy for the following graph.

- The starting node is S and the goal node is G.

- Heuristic values are given next to each node.

- SA comes before SB.
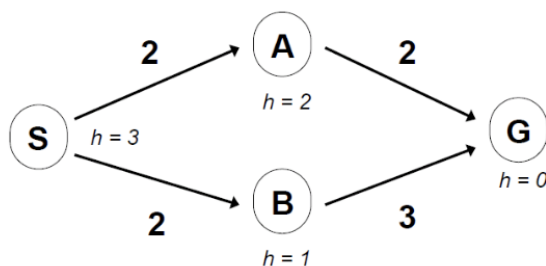


**Note:** This small example only show the inner working of A*. It does not demonstrate its advantage over LCFS.
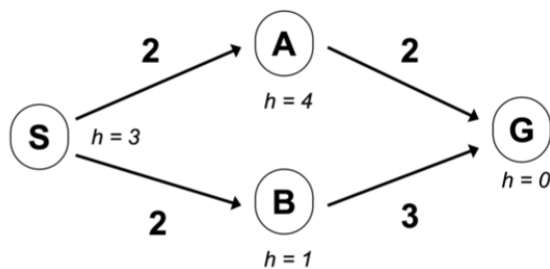
**heuristic function**
h(S) = 3
h(A) = 2
h(B) = 1
h(G) = 0

Answer:

```
+ S,3       # 0 + 3 = 3
- S,3
+ SA,4      # 2 + 2 = 4
+ SB,3      # 2 + 1 = 3
- SB,3
+ SBG,5     # 5 + 0 = 5
- SA,4
+ SAG,4     # 4 + 0 = 4
- SAG,4
```

13

**heuristic function**
h(S) = 3
h(A) = 4
h(B) = 1
h(G) = 0

- Same example as the one before just assume *h(A) = 4* instead.



Answer:

```
+ S,3      # 0 + 3 = 3
- S,3
+ SA,6     # 2 + 4 = 6
+ SB,3     # 2 + 1 = 3
- SB,3
+ SBG,5    # 5 + 0 = 5
- SBG,5
```

**Non-optimal solution! Why?**

# A*: proof of optimality

When using A* (without pruning) the first path *p* from a starting node to a goal node that is selected and removed from the frontier has the lowest cost.

Sketch of proof:

- Suppose to the contrary that there is another path from one of the starting nodes to a goal node with a lower cost.

- There must be a path *p*' on the frontier such that one of its continuations leads to the goal with a lower overall cost than *p*.

- Since *p* was removed before *p*':

$$f(p) \leq f(p') \implies cost(p) + h(p) \leq cost(p') + h(p') \implies cost(p) \leq cost(p') + h(p')$$

- Let *c* be any continuation of *p*' that goes to a goal node; that is, we have a path *p*'*c* from a start node to a goal node. Since *h* is admissible, we have:

$$cost(p'c) = cost(p') + cost(c) \geq cost(p') + h(p')$$

- Thus:

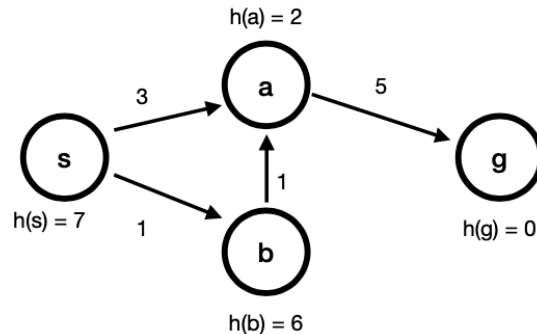$$cost(p) \leq cost(p') + h(p') \leq cost(p') + cost(c) = cost(p'c)$$

16

# Effect of pruning on A*

Trace the frontier in A* search for the following graph, with and without pruning.

```
nodes={s, a, b, g},
estimates = {s:7, a:2, b:6, g:0},
edge_list=[(s,a,3), (s,b,1),
           (b,a,1), (a,g,5)],
starting_nodes = [s],
goal_nodes = {g}.
```



### Answer <u>without</u> pruning

```
+ S,    7
- S,    7
+ SA,   5
+ SB,   7
- SA,   5
+ SAG,  8
- SB,   7
+ SBA,  4
- SBA,  4
+ SBAG, 7
- SBAG, 7
```

### Answer **with** pruning

```
# expanded={}
+ S,    7
- S,    7        # expanded={S}
+ SA,   5
+ SB,   7
- SA,   5        # expanded={S,A}
+ SAG,  8
- SB,   7        # expanded={S,A,B}
+ SBA,  4!
- SAG,  8       Non-optimal solution!
```

17

## What went wrong when pruning $A$ Search

- An expensive path, *sa* was expanded before a cheaper path *sba* could be discovered, because $f(sa) < f(sb)$
- Is the heuristic function *h* admissible?

    - Yes

- So what can we do?

    - We need a stronger condition than admissibility to stop this from happening

> Principle: When we are removing nodes, we are essentially saying we have found a cheaper solution, in this case, this was not true and hence why the algorithm fails, we need to use a stronger condition as outlined below

## Monotonicity

A heuristic function is monotone or consistent if for every two nodes $n$ and $n'$ which is reachable from $n$:

$$h(n) \leq cost(n, n') + h(n')$$

With the monotone restriction, we have:

$$
\begin{aligned}
f(n') &= cost(s, n') + h(n') \\
&= cost(s, n) + cost(n, n') + h(n') \\
&\geq cost(s, n) + h(n) \\
&\geq f(n)
\end{aligned}
$$

How about using the actual cost as a heuristic?

- Would it be a valid heuristic?
- Would we save on nodes expanded?
- What's wrong with it?

    - It becomes as computationally expensive as it is to just do the problem

Choosing a heuristic: a trade-off between quality of estimate and work per node!

# Dominance relation

- Dominance: $h_a \geq h_c$ if
$$\forall n : h_a(n) \geq h_c(n)$$

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible
$$h(n) = max(h_a(n), h_b(n))$$

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic

$exact$
$|$
$max(h_a, h_b)$

$h_a \qquad h_b$

$|$
$h_c$

$zero$
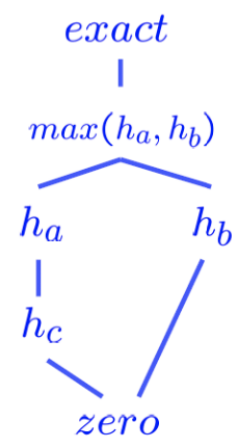
24

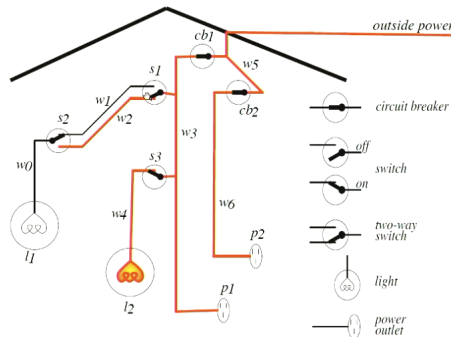**Figure 12:** Dominance Relation

> Further algorithms are discussed in this segment of the and lecture *boarders onto lecture three* however this content will not be assessed in the duration of this course.

**Lecture Three: Knowledge Base and Information**

**How to represent information in a knowledge base**

Information                                                    Knowledge Base



- The computer doesn't know the meaning of the symbols (logical and etc...)
- The user can interpret the symbol using their meaning
- There is no specific syntax for this, it is just what ever is readable for the user/writer

**Simple language and definitions**

- An **atom** is a symbol starting with a lower case letter
- A **body** is an atom or is of the form $b_1 \wedge b_2$ where $b_1$ and $b_2$ are bodies
- A **definite clause** is an atom or a rule of the form $h \leftarrow b$ where $h$ is an atom and $b$ is a body
- A **knowledge base** is a set of definite clauses
- An **interpretation** $i$ assigns a truth value to each atom
- A **body** $b_1 \wedge b_2$ is true in $i$ if $b_1$ is true in $i$ and $b_2$ is true in $i$
- A **rule** $h \leftarrow b$ is false in $i$ if $b$ is true in $i$ and $h$ is false in $i$, the rule is true otherwise
- A **knowledge base** $KB$ is true in $i$ if and only if every clause in $KB$ is true in $i$
- A **model** of a set of clauses is an interpretation in which all the clauses are *true*
- If $KB$ is a set of clauses and $g$ is a conjunction of atoms, $g$ is a **logical consequence** of $KB$, this is denoted as $KB \models g$, if $g$ is *true* in every model of $KB$

  – That is, $KB \models g$ if there is no interpretation in which $KB$ is *true* and $g$ is *false*.

- A **Proof procedure** is a -possibly non-deterministic - algorithm for deriving consequences of a knowledge
- Given a proof procedure, $KB \vdash g$ means $g$ can be derived from knowledge base $KB$
- Recall $KB \models g$ means $g$ is *true* in all models of $KB$
- A proof procedure is **sound** if $KB \vdash g \implies KB \models g$
- A proof procedure is **complete** if $KB \models g \implies KB \vdash g$

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

How many interpretations?

|       | p     | q     | r     | s     | model of KB? |
|-------|-------|-------|-------|-------|--------------|
| $I_1$ | true  | true  | true  | true  |              |
| $I_2$ | false | false | false | false |              |
| $I_3$ | true  | true  | false | false |              |
| $I_4$ | true  | true  | true  | false |              |
| $I_5$ | true  | true  | false | true  |              |

Which of $p, q, r, s$ logically follow from KB?

**Figure 13:** simple example question

Answers to the questions:

> We have four atoms {p,q,r,s}, because we have 4 atoms, there are 16 permutations in our truth table ($2^4$), therefore we have 16 interpretations

**Bottom-up proof procedure**

Rule of derivation:

if $h \leftarrow b_1 \wedge ... \wedge ...b_m$ is a clause in the knowledge base, and each $b_i$ has been derived, then $h$ can be derived

- This is **Forward chaining** on this clause (this rule also covers the case when $m = 0$)
- $KB \vdash g$ if $g \in C$ at the end of the below algorithmic procedure

- Tracing tutorial: 1:13:30

$$C := \{\};$$
**repeat**
$\qquad$ **select** clause "$h \leftarrow b_1 \wedge \ldots \wedge b_m$" in $KB$ such that
$\qquad\qquad b_i \in C$ for all $i$, and
$\qquad\qquad h \notin C;$
$\qquad C := C \cup \{h\}$
**until** no more clauses can be selected.

**Figure 14:** Bottom-up proof procedure algorithm pseudo code

**Top-down proof procedure**

Idea: search backward from a query to determine if it is a logical concequence of $KB$

An **answer clause** is of the form:

- $yes \leftarrow a_i \wedge \ldots \wedge a_m$

The SLD Resolution of this answer clause on atom $a_i$ with the clause:

- $a_i \leftarrow b_1 \wedge \ldots \wedge b_p$
- Tracing tutorial: 1:31:00

An **answer** is an answer clause with $m = 0$. That is the answer clause $yes \leftarrow$.

A **Derivation** of query $?q_1 \wedge \ldots \wedge q_k$ from $KB$ is a sequence of answer clauses $\lambda_0, \lambda_1, \ldots \lambda_n$

- $\lambda_0$ is the answer clause $yes \leftarrow q_1 \wedge \ldots \wedge q_k$
- $\lambda_1$ is obtained by resolving $\lambda_{i-1}$ with a clause in $KB$
- $\lambda_n$ is the answer

To solve the query $?q_1 \wedge \ldots \wedge q_k$:

$\quad ac := \text{``yes} \leftarrow q_1 \wedge \ldots \wedge q_k\text{''}$

$\quad$**repeat**

$\quad\quad\quad$**select** atom $a_i$ from the body of $ac$

$\quad\quad\quad$**choose** clause $C$ from $KB$ with $a_i$ as head

$\quad\quad\quad$replace $a_i$ in the body of $ac$ by the body of $C$

$\quad$**until** $ac$ is an answer.

**Figure 15:** Top-down proof procedure algorithm pseudo code

> There is more information on SLD Resolution at the end of this lecture, this will be needed in the assignment

**Lecture Four: Declarative Programming**

**What is declarative programming?**

Declarative programming is the use of mathematical logic to describe the logic of computation without describing its control flow

- Knowledge bases and queries in propositional logic are made up of propositions and connectives
- Predicate logic adds the notion of *predicates* and *variables*
- We take a non-theoretical approach to predicate logic by introducing *declarative programming*
- useful for: expert systems, diagnostics, machine learning, parsing text, theorem proving, …

**Datalog**

- Prolog is a declarative programming language and stand for PROgramming in LOGic
- we only look at a sybset of the language which is equal to Datalog
- Think declaratively, not procedurally
- High level, interpreted language
- We will have a file that contains a knowledge base, and we will have an interpreter where we can ask queries

Here is an example of a knowledge base in Datalog:

```
1  woman(mia)
2  woman(jody)
3  woman(yolanda)
4  playesAirGuitar(yolanda)
```

Here is how we may query data using the interpreter:

```
1  $ woman(mia)
2  yes
```

> Further examples of this are in the slides of lecture four

Operators

- Implication :-
- Conjunction: , (AND)
- Disjunction ; (OR)
- We will later talk about how to simulate the (NOT) operator

Interpreter Operands and rules:

- Variables: X, Y, Z, Cam, AnythingThatStartswithUppercase

    - Acts as a `wildcard` to match with when querying

- Order of arguments matters
- `Arity` is important
- Unification/matching:

    - Two terms unify or match if they are the same term or if the contain variables that can be uniformly instanciated with terms in such a way that the resulting terms are equal (this is how we query)
    - Example: $l(s(g), Z) = k(X, t(Y))$

With only Unification we can do some programming

```
1  vertical(line(point(X,Y), point(X,Z))
2  horizontal(line(point(X,Y), point(Z,Y))
```

**Proof Search**

- Prolog has a specific way of answering queries

    - Search knowledge base from top to bottom
    - Processes clauses from left to right

- Backtracking to recover from bad choices

- Further examples using prolog: 1:10:00

**Recursive Programming**

```
1  child(anna, bridget)
2  child(bridget, caroline)
3  child(caroline, donna)
4  child(donna, emily)
5  decendent(X,Y):-child(X,Y)
6  decendent(X,Y):-child(X,Z), decendent(Z,Y)
```

If we make the following query with the above knowledge base, we get a positive response

```
1  $- decendent(anna, donna)
2  yes
```