Harry Langford
hjel2@cam.ac.uk

Part I

Sean's Exercise Sheet Part 1

3 Search

1. Explain why breadth-first search is optimal if path-cost is a non-decreasing function of node depth.

   Breadth-first search searches in order of depth; if the first goal node found by breadth-first search n_g is found at depth d_g then any goal nodes n'_g found later will be found at depth $d'_g \geq d_g$.

   Since path-cost is a non-decreasing function of node depth, we have for some non-decreasing function f, $p(n_g) = f(d_g) \leq f(d'_g) = p(n'_g)$. So $p(s_g) \leq p(s'_g)$. Therefore, the first goal node will have a lower path cost than any goal nodes found later. Hence breadth-first search is optimal under the assumption that path-cost is a non-decreasing function of node depth.

2. In the graph search algorithm, assume a node is taken from the fringe and found *not* to be a goal and *not* to be in `closed`. We then add it to `closed` and add its descendants to `fringe`.

   (a) When might it make sense to add *all* descendants to the queue without checking if they are in `closed` first?

   It would make sense to add all descendants to the queue without checking if they are in `closed` if we knew the graph we were searching was a tree. In a tree there is only one path from the root to any node. So we would know that none of the descendants would be in `closed` – making it a redundant test.

   We may implement the fringe as a fibonacci heap – or some other structure for which we cannot actually check in constant time.

   We are going to check when we remove the node from the fringe – so we're already going to check it anyway. It's easiest to just avoid checking on insertion and minimize the total amount of work.

   Alternatively, we could have a set. Then test before entering and call decreasekey on the node in the fibonacci heap. This can be more expensive if you have a high branching factor (high connectivity).

   (b) Explain why the code in the lecture notes has been changed to add descendants to the queue only if they are not already in `closed`.

   The code in the lecture notes is a *graph* search algorithm – it aims to work on all graphs. So should not have problem-specific optimisations.

   If this check was not present, searches on undirected (or cyclic) graphcs would no longer be complete. For example a DFS on an undirected graph would loop infinitely on the first two nodes.

3. Iterative deepening depends on the fact that *the vast majority of the nodes in a tree are in the bottom level.*

   This is proving that iterative deepening isn't mad – and that throwing away all this work makes sense. This is dominated by a

   - Denote by $f_1(b, d)$ the total number of nodes appearing in a tree with branching factor b and depth d. Find a closed-form expression for $f_1(b, d)$.

$$\begin{aligned} f_1(b,d) &= \sum_{i=0}^{d} b^i \\ &= \frac{b^{d+1}-1}{b-1} \end{aligned}$$

- Denote by $f_2(b,d)$ the total number of nodes generated in a complete iterative deepening search to depth d of a tree having branching factor b. Find a closed-form expression for $f_2(b,d)$ in terms of $f_1(b,d)$.

$$\begin{aligned} f_2(b,d) &= \sum_{i=0}^{d} f_1(b,i) \\ &= \sum_{i=0}^{d} \frac{b^{i+1}-1}{b-1} \\ &= \frac{1}{b-1}\left(\sum_{i=0}^{d} b^{i+1} - 1\right) \\ &= \frac{1}{b-1}\left(\left(b\sum_{i=0}^{d} b^{i+1}\right) - d - 1\right) \\ &= \frac{1}{b-1}\left(b\frac{b^{d+1}-1}{b-1} - d - 1\right) \\ &= \frac{b^{d+2} - 2 * b - bd + d + 1}{(b-1)^2} \end{aligned}$$

- How do $f_1(b,d)$ and $f_2(b,d)$ compare when b is large?

  For large b, $f_1(b,d) \approx f_2(b,d)$.

  This is $f_2 = \frac{b+1}{b} \cdot f_1 + c$.

  We run out of RAM before you run out of time because "you can always wait longer but if you run out of memory the algorithm will fail"

4. The $A^\star$algorithm does not perform a goal test on any state *until it has selected it for expansion.* We might consider a slightly different approach: namely, each time a node is expanded check all of its descendants to see if they include a goal.

   Give two reasons why this is a misguided idea, where possible illustrating your answer using a specific example of a search tree for which it would be problematic.

   - Under the alternative algorithm, we no longer test nodes in ascending order of estimated cost. This means, *even with monotonic heuristics* the algorithm will not be optimal. The first node we test may be a highly suboptimal goal state or have been found by a highly suboptimal path.

     Consider the graph below:

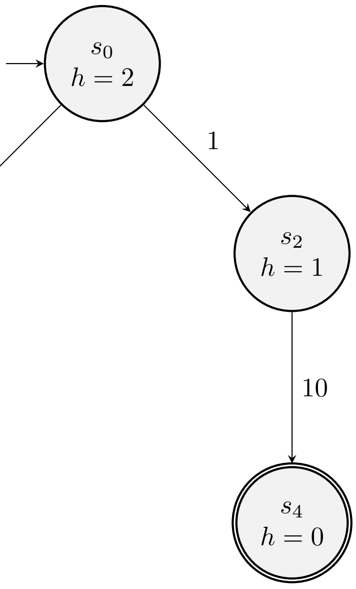


Figure 1: Graph on which alternative A* finds a non-optimal goal

In this example, the heuristic is both admissible and monotonic. However, since it underestimates the cost of s_2 more than s_1, s_2 is explored and expanded first, leading to a highly suboptimal goal.

The last hop isn't taken into account, so it's not optimal. Sometimes this doesn't matter. But you've got to do $b-1$ times as many goal checks. If goal checks are simple then it may not be a problem (although it's still worse than the normal algorithm).

But if goal checks are complicated, this may become intractable.

"push work as far back as possible"

- Consider an example where testing for a goal state is computationally expensive (i.e involves running a simulation) – but we have a very good monotonic heuristic. If the graph has a high branching factor, then the cost of searching the graph becomes dominated by unnecessary tests for goal states – this "optimisation" increases compute-time by a factor of $b-1$.

5. The f-cost is defined in the usual way as

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

where n is any node, p denotes path cost and h denotes the heuristic. An admissible heuristic is one which, for any n

$$h(n) \leq \text{actual distance from } n \text{ to the goal}$$

and a heuristic is monotonic if for consecutive nodes n and n' it is always the cast that

$$f(n') \geq f(n)$$

- Prove that h is monotonic if and only if it obeys the triangle inequality, which states that for any consecutive nodes n and n'

$$h(n) \leq c_{n \to n'} + h(n)$$

where $c_{n \to n'}$ is the cost of moving from n to n'.

Assume h is monotonic:

$$\begin{aligned} f(n) &\leq f(n') \\ p(n) + h(n) &\leq p(n') + h(n') \\ p(n) + h(n) &\leq p(n) + c_{n\to n'} + h(n') \\ h(n) &\leq c_{n\to n'} + h(n') \end{aligned}$$

Therefore, if h is monotonic, then it obeys the triangle inequality.

Consider also the case when the shortest path to n' doesn't go through n.

Case split midway through this expression. If n isn't the case then the path cost to n' is even greater. So this inequality is "even more unequal".

Assume h obeys the triangle inequality:

$$\begin{aligned} h(n) &\leq c_{n\to n'} + h(n') \\ p(n) + h(n) &\leq p(n) + c_{n\to n'} + h(n') \\ p(n) + h(n) &\leq p(n') + h(n') \\ f(n) &\leq f(n') \end{aligned}$$

Therefore, if h obeys the triangle inequality then it is monotonic.

Since we have proved both directions, we have that h is monotonic if and only if it obeys the triangle inequality.

- Prove that if a heuristic is monotonic then it is also admissible

  Assume a heuristic is monotonic.

  Next, assume there is a path $n_0 \to n_1 \to \cdots \to n_k$ for some node n_0 and some goal node n_k. Using monotonicity, we have:

$$\begin{aligned} f(n_0) &\leq f(n_1) \leq \cdots \leq f(n_k) \\ f(n_0) &\leq f(n_k) \\ p(n_0) + h(n_0) &\leq p(n_0) + h'(n_0) \\ h(n_0) &\leq h'(n_0) \end{aligned}$$

  This is the admissibility property. Therefore, if a heuristic is monotonic then it is also admissible.

  Any valid heuristic must have $h(goal) = 0$

  The properties of monotonicity only holds on the **shortest path**. Properties which aren't on the shortest path don't matter since $A^\star$only considers the shortest path.

- Is the converse true? Either prove that this is the case or provide a counterexample.

  The converse is not true. I provide an example where the heuristic is admissible, but not monotonic.

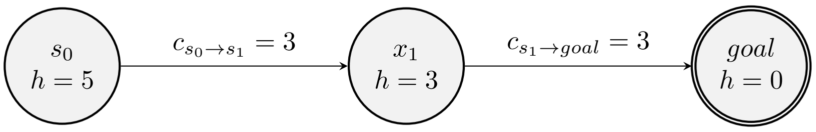


Figure 2: Counterexample to Admissibility implying Monotonicity

In the first edge, the real cost decreases faster than the heuristic.

In the second edge, the real cost goes down slower than the heuristic.

7. In some problems we can simultaneously search:

   - *forward* form the *start* state
   - *backwards* from the *goal* state

   until the searches meet. This seems like it might be a very good idea:

   - If the search methods have complexity $\mathcal{O}\left(b^d\right)$ then...
   - ...we are converting this to $\mathcal{O}\left(2b^{d/2}\right) = \mathcal{O}\left(b^{d/2}\right)$.

   (Here, we are assuming the branching factor is b in both directions). Why might this not be as good an idea as it seems?

   In the general case, the start state is not known in advance. This may make a "good" heuristic for the distance *to* the start state impossible to automatically generate.

   Furthermore, even if we were to make such a heuristic; there is no guarantee that the forward and reverse seearches would meet at an optimal node. In-fact it would be impossible (in the general case) to even prove a *bound* on how suboptimal the solution we found was.

   Therefore, the algorithm would require human intervention (to make a good heuristic) and would no longer give an optimal path cost – or a near-optimal path cost.

   There are obviously edge-cases where one or both of these problems do not apply.

   Other reasons include:

   - There may be multiple goals
   - The branching factor in reverse may be far higher
   - The reverse heuristic may not exist at all! States may be hard to undo.
   - This completely fails in the presence of fluent properties

8. Suggest a method for performing depth-first search using only $\mathcal{O}(d)$ space.

   The algorithm I provide is designed to search a graph in $\mathcal{O}(d)$ space. Searching a tree in $\mathcal{O}(d)$ space is trivial. This algorithm works by considering the graph as a tree – this means it may revisit nodes.

```
def dfs(n: node, seen: set[node]):
  if goal(node):
    return node
  for neighbour in node.neighbours:
    if neighbour not in seen:
      dfs(neighbour, seen | {neighbour})
```

   Sometimes, you accept you're going to have do redo work – "remembering all the work I've done is larger than the amount of RAM I have".

   So we will redo work. Provided the graph is weakly connected, it may be more efficient to assume all work is new *even if* it may not be.

   The argument that "this is $\mathcal{O}(d)$" (instead of $\mathcal{O}(d \lg b)$) works if the graph is in memory and you use pointers. You can't handle more data than $\mathcal{O}(b)$ so it somewhat makes sense.

9. One modification to the basic local search algorithm suggested in the lectures was to make steps probabilistically, but only if the value of f is *improved*.

   (a) Would it be a good idea to also allow steps that move to a state with a *worse* value for f?

"Local Search Algorithms" use only local knowledge to try and find a global maximum.

Yes. In cases where the function state space f is not well-behaved (contains local maxima), we may be able to find a more optimal solution if we explore parts of the graph appear to be slightly worse in the short-term. However, under this strategy we may wish to remember the best state we have found in-case this exploration does not yield a better state.

(b) Suggest an algorithm for achieving this such that you have some control between steps that increase or decrease f.

I propose an algorithm similar to simulated annealing:

```
def anneal($s_0$, f, $t_{\max}$, $\delta$):
    $s$ = $s_0$
    $t$ = $t_{\max}$
    best = $s_0$
    while True:
        best = max(best, max($s$.neighbours, key=f), key=f)
        $w$ = [$f(s')$ + $t$ - $f(s)$ for $s'$ in $s$.neighbours]
        $p$ = [$f_{s'}$ / sum($w$) for $s'$ in $w$ if $f_{s'}$ > 0]
        if not p:
                return best
        $s$ = random.choice($s$.neighbours, $p$)
        $t$ -= $\delta$
```

Shaun Holden recommends allocating a certain probability to increasing steps and a certain probability to worsening probability.

Assign a uniform probability to every worsening step.

Gradually decrease the probability which is allocated to worsening steps.

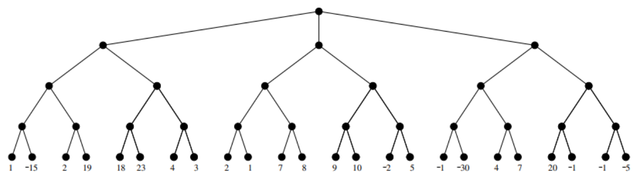
The probabilities are arbitrary and hard to generalise.

Part II

Sean's Exercise Sheet Part 2

1 Games

1. Consider the following game tree:



Large outcomes are beneficial for Max. How is the tree pruned by $\alpha - \beta$ minimax if Max moves first? How is it pruned if Min is the root, and therefore moves first.

Using i, j as an abbreviation for $\alpha = i$, $\beta = j$:

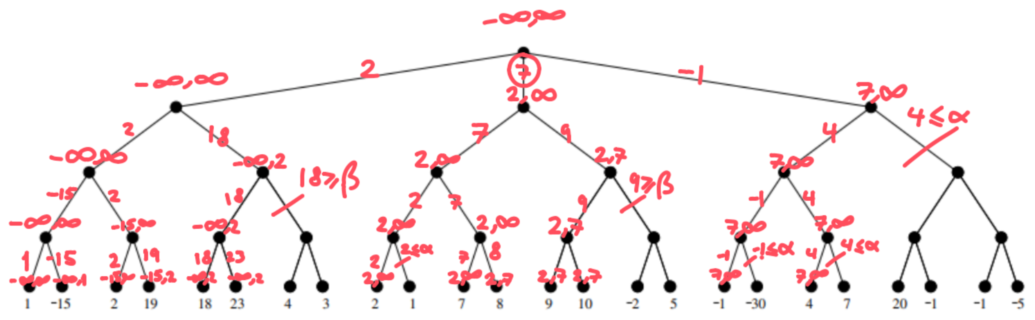


Figure 3: Max achieves 7

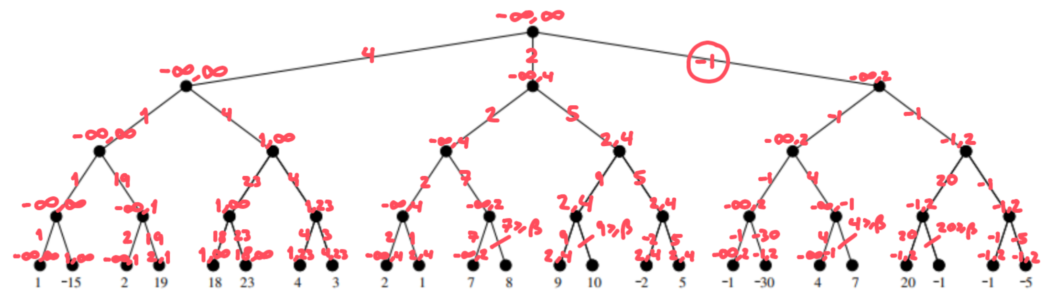


Figure 4: Min achieves -1

This is wrong! The 6th node from the right should also be pruned.

This works better if you prune the best nodes first. You get different answers if you explore the nodes in different orders.

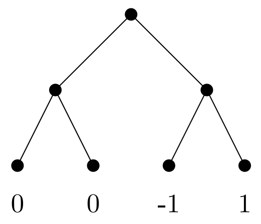
3. Is the minimax approach to playing games optimal against an imperfect opponent? Either prove this is the case or give a counterexample.

   "Without knowing your your opponent will play, you can't maximise your score".

   This depends on our definition of "optimal"; and in what way the opponent is imperfect. If the opponent has a nonzero probability of making any move and our definition of "optimal" is "the move which maximises the worst-case outcome", then minimax is optimal.

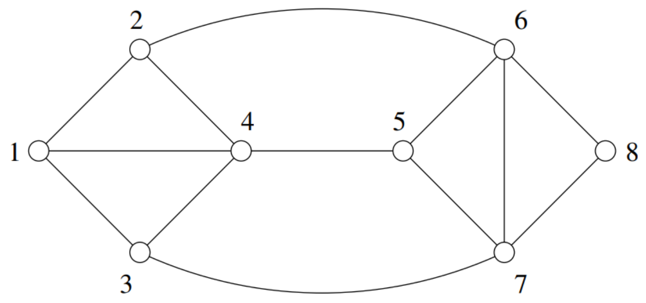
   If our definition of "optimal" is "maximising expected utility" then minimax may not be optimal.

   Additionally, if our opponent has a zero probability of making some moves (i.e because of an affection heuristic, they never move left on binary branches) then minimax is not optimal. In the figure below, minimax would find a solution with utility 0 for Max. If we exploit Min's affection heuristic, Max could achieve a utility of 1.



2 Constraint Satisfaction Problems

1. Consider the following constraint satisfaction problem:



We want to colour the nodes using the colours red (R), cyan (C) and black (B) in such a way that connected nodes have different colours.

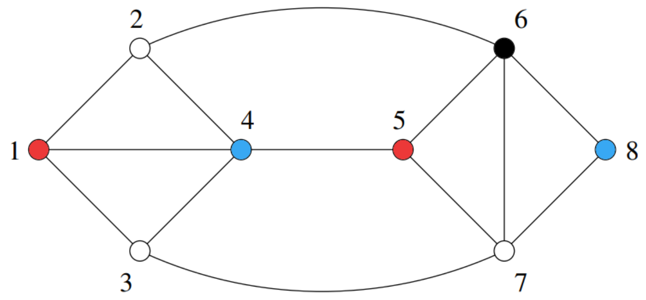
- Assuming we attempt the assignments $1 = R$, $4 = C$, $5 = R$, $8 = C$, $6 = B$. Explain how *forward checking* operates in this example, and how it detects the need to backtrack.

  In forward-checking, after assigning a node a value, we remove this value from the domain of all adjacent nodes. An partial instantiation I_k of k variables is said to conflict with node V_i if, under forward checking V_i has no valid assignments. Once any node conflicts with the current partial initialisation, partial checking will determine that the search cannot succeed and will backtrack.

  Under forward checking (after making the first 4 assignments) the graph would be in the following state:

node	assigned	values
1	R	$\{R\}$
2		$\{B\}$
3		$\{B\}$
4	C	$\{C\}$
5	R	$\{R\}$
6		$\{B\}$
7		$\{B\}$
8	C	$\{C\}$

  Once the search attempted to assign $6 = B$, it would remove B from the possible values of all adjacent nodes. This would leave the graph in the following state:



node	assigned	values
1	R	$\{R\}$
2		$\{\}$
3		$\{B\}$
4	C	$\{C\}$
5	R	$\{R\}$
6	B	$\{B\}$
7		$\{\}$
8	C	$\{C\}$

On assigning $6 = B$, forward checking would check that this was consistent with some value on all adjacent nodes. It would then backtrack to $8 = C$ and try $8 = B$, continuing the search.

- Will the AC-3 algorithm detect a problem earlier in this case? Explain the operation of the algorithm in the example.

  The AC-3 algorithm would detect a problem earlier in this case. The AC-3 algorithm is a way of implementing constraint propogation. It ensures that for all edges $V_i \to V_j$, V_i is consistent with V_j. This means that for all assignments of V_i, there exists an assignment to V_j which is consistent with that. If this is not the case, AC-3 will remove this value from the set of possible values of V_i – and then recursively check all edges $V_k \to V_i$ for consistency.

  After assigning $1 = R$, the possible values are as follows:

node	assigned	values
1	R	$\{R\}$
2		$\{B, C\}$
3		$\{B, C\}$
4		$\{B, C\}$
5		$\{R, B, C\}$
6		$\{R, B, C\}$
7		$\{R, B, C\}$
8		$\{R, B, C\}$

After assigning $4 = C$, the possible values (under AC-3) are as follows:

node	assigned	values
1	R	$\{R\}$
2		$\{B\}$
3		$\{B\}$
4	C	$\{C\}$
5		$\{R, B, C\}$
6		$\{R, C\}$
7		$\{R, C\}$
8		$\{R, B, C\}$

After assigning $5 = R$, the possible values are as follows:

node	assigned	values
1	R	$\{R\}$
2		$\{B\}$
3		$\{B\}$
4	C	$\{C\}$
5	R	$\{R\}$
6		$\{C\}$
7		$\{C\}$
8		$\{R, B\}$

AC-3 has determined that C is not a consistent value for 8 to take. So AC-3 notices that the current assignment is inconsistent earlier and backtracks earlier.

Think of AC-3 as "propagating removals". Rather than assigning a value – you should be removing values from their set of possible values. Then propagate this to all neighbours.

Part III
Exam Questions

1 2016 Paper 4 Question 2

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2016p4q2.pdf

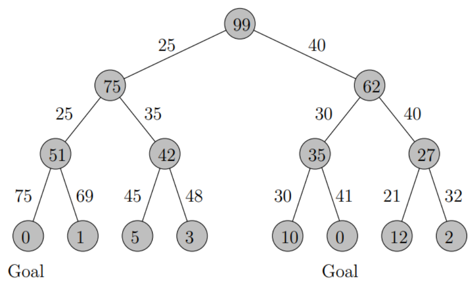
(a) Provide a detailed description of the Iterative Deepening A* (IDA*) algorithm. Your answer should include a clear statement of the algorithm in pseudo-code, and a general description of how it works.

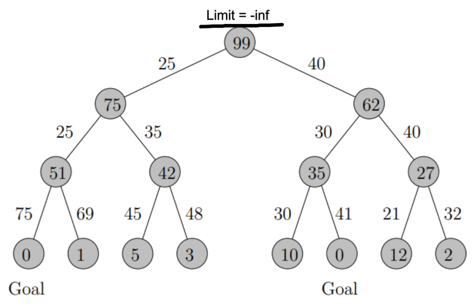
In the IDA*algorithm, we repeatedly perform depth-first searches (bounded by f-value of nodes). After each failed attempt, we increase the bound on f-value to the least f-value we discovered which was beyond the previous bound. This is repeated until any goal state is found. This goal state is guaranteed to be optimal under the same conditions for which $A^\star$search is optimal.

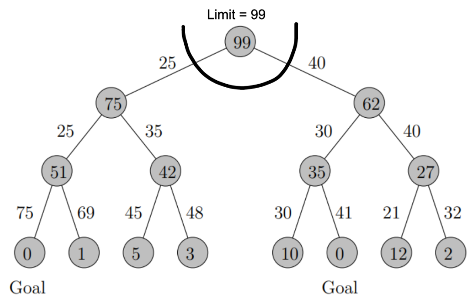
```
def contour($s$, limit, path):
        if f($s$) > limit:
                return [], f($s$)
        if goal($s$):
                return path + [$s$], f($s$)
        next_f = $\infty$
        for $s'$ in $s$.edges:
                newPath, f = contour($s'$, limit, path + [$s$])
                if newPath:
                        return newPath, f
                next_f = min(f, next_f)
        return [], next_f

def IDA*($s_0$):
        if goal($s_0$):
                return [root], f($s_0$)
        path, limit = [], f($s_0$)
        while not path and limit != inf:
                path, limit = contour($s_0$, limit, [$s_0$]
        return path
```

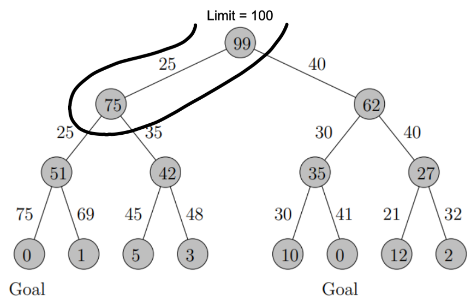
(b) Explain how the IDA*algorithm searches the following search tree. Numbers between nodes denote the cost of the path between those nodes, and numbers on the nodes denote the value of the heuristic for that node.

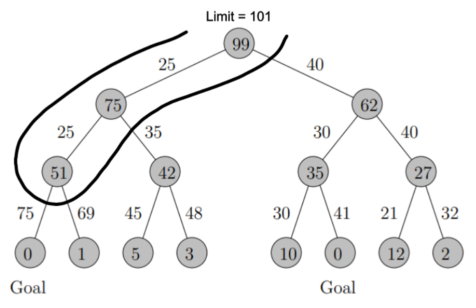
Start off with the limit set to the f-cost of the root.

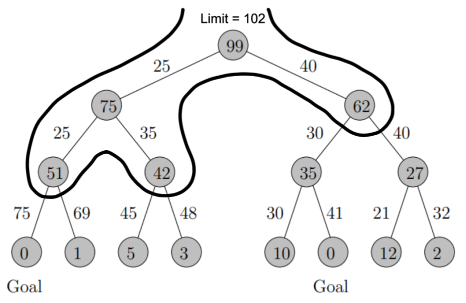


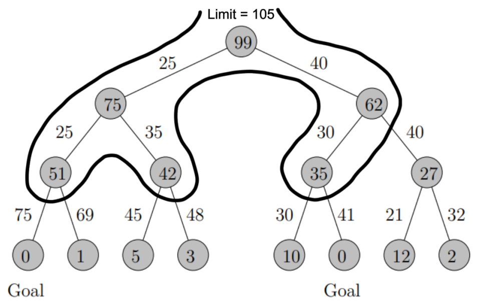


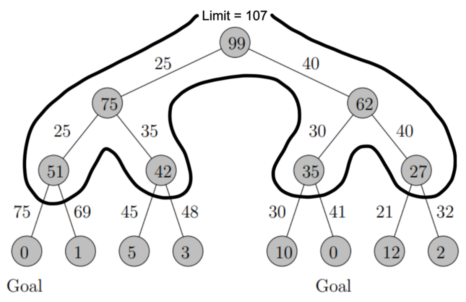


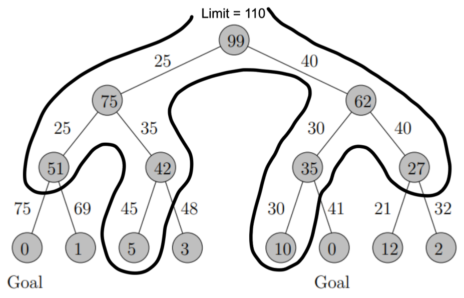


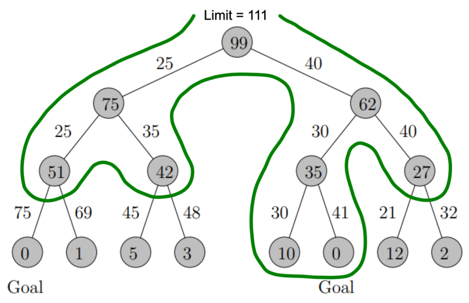

Limit = 101
99
25
40
75
62
25
35
30
40
51
42
35
27
75
69
45
48
30
41
21
32
0
1
5
3
10
0
12
2
Goal
Goal


Limit = 102
Goal
Goal


Limit = 105
Goal
Goal


Limit = 107
Goal
Goal


Limit = 110
Goal
Goal



(c) Give *two* reasons why the IDA* algorithm might prove unsuitable as a solution to a search problem, in each case giving a brief explanation of why this is the case and suggesting a potential solution.

If either of the following conditions are met then IDA*may fail to find any solution:

- Edges have no minimum cost.

  In this case, IDA*can keep exploring a path which tends to a value lower than that of the optimum solution. We can solve this by constraining all edges to have a cost $\geq \varepsilon$ for some arbitrarily small ε.

- Nodes can have an infinite number of neighbours.

  In this case, once a node is expanded, the search algorithm will keep checking all of its neighbours. In the case that none of them are goal states, this leads to an infinite search.

  In this case, the problem is not suited to a tree-search approach. There may be some domain-specific solutions; however there is no general approach which makes this solvable.

In the following two cases, IDA*will not be efficient:

- nodes have unique f-values

  In this case, every successive call to IDA*will search *one more node*. This means the time required to find a solution is quadratic in the number of nodes with f-value smaller than it. In large trees, this makes the problem almost intractable.

  This can be solved by increasing the f-limit by some arbitrarily small ε, yielding a goal state which is guaranteed to have a cost within ε of the optimal goal state.

  You can solve by:

  – increasing by ε

  – rounding heuristics to the nearest ε

  – Combining IDA*and Iterative Deepening. Search to the minimum of the depth or f-value + ε. You add at most one layer to the tree.

  "force the search to make some progress each time"

- too many nodes share the same f-value

  In this case, successive calls to IDA*will dramatically increase the size of the tree – rendering the search process incredibly inefficient and similar to a complete search.

- This heuristic has to be monotonic!

- You may not be able to think of a heuristic.

2 2017 Paper 4 Question 1

https://www.cl.cam.ac.uk/teaching/exams/pastpapers/y2017p4q1.pdf

`EvilRobot` has two dogs called `Fido` and `Fifi`. All three of them enjoy `pie` and `sausages` so much that they like to steal them. At the beginning of the day the `butcher` has some `sausages` and the `pieShop` has some `pie`. Also, `EvilRobot` and his pets are at home, but they aim to end the day having relieved the local businesses of their products.

(a) Give a detailed definition of a *Constraint Satisfaction Problem (CSP)*. Include in your answer a definition of what it means for an assignment to be *consistent* and to be *complete*, and for an assignment to be a *solution*.

A constraint satisfaction problem is the problem of assigning values from domains $D_1, \dots, D_n$ to a set of variables $V_1, \dots, V_n$ such that all constraints $C_1, \dots C_m$ are met.

Constraints are sets of values which variables can take.

Variables take values from separate domains.

An assignment is consistent if it violates no constraints. An assignment is complete if it assigns a value to every variable. An assignment is a solution if it is both consistent and complete.

An assignment is consistent if "every checkable constraint is satisfied".

You have three types of constraints – those which are satisfied, those which are violated and those which can't be checked because variables in the constraint aren't satisfied.

Constraints which "can't be checked" are where interesting algorithms arise.

(b) Consider the constraint C on four variables $\{V_1, V_2, V_3, V_4\}$ each of which has the domain `{true, false}`, with

$$\begin{aligned}C = \{&(\texttt{true}, \texttt{true}, \texttt{true}, \texttt{true})\\ &(\texttt{true}, \texttt{false}, \texttt{true}, \texttt{false})\\ &(\texttt{false}, \texttt{true}, \texttt{false}, \texttt{false})\\ &(\texttt{false}, \texttt{false}, \texttt{false}, \texttt{true})\\ &(\texttt{false}, \texttt{false}, \texttt{true}, \texttt{true})\}\end{aligned}$$

Explain how this constraint can be replaced by a collection of binary constraints having an identical effect.

This constraint can be replaced by a collection of binary constraints by introducing an auxiliary variable A which takes a value from $\{1, 2, 3, 4\}$ and is i if the i^{th} formula is being satisfied.

You want conditions of the form $A \to V_i = true$ etc etc.

A binary constraint is a constraint of the form $C = \{(a, b), (c, d), (e, f), (g, h), (i, j) \dots\}$ – every tuple of acceptable values has at most two arguments.

Changing to this simplifies the format that the input solver needs to handle.

$$\begin{aligned}C = \{&\{(A=0, V_1=\texttt{true}), (A=1, V_1=\texttt{true}), (A=2, V_1=\texttt{false}), (A=3, V_1=\texttt{false}), (A=4, V_1=\texttt{false})\},\\ &\{(A=0, V_2=\texttt{true}), (A=1, V_2=\texttt{false}), (A=2, V_2=\texttt{true}), (A=3, V_2=\texttt{false}), (A=4, V_2=\texttt{false})\},\\ &\{(A=0, V_3=\texttt{true}), (A=1, V_3=\texttt{true}), (A=2, V_3=\texttt{false}), (A=3, V_3=\texttt{false}), (A=4, V_3=\texttt{true})\},\\ &\{(A=0, V_4=\texttt{true}), (A=1, V_4=\texttt{false}), (A=2, V_4=\texttt{false}), (A=3, V_4=\texttt{true}), (A=4, V_4=\texttt{true})\}\end{aligned}$$

(c) Describe the state-variable representation for planning problems. Illustrate your answer by showing how the action of `EvilRobot` (or one of his pets) stealing something could be represented in the scenario set out at the beginning of this question.

(d) Explain how the *state* of a planning problem can be represented in the state-variable representation.

(e) Using your example of the stealing action provided in part (c), explain how this planning problem might be translated into a CSP. You should include in your explanation examples of the translation for actions, state variables, and action preconditions and action effects, but you need not describe the translation for frame axioms.

These problems only work on very specific use-cases!

In classical AI, you have to know so much about the problem you're trying to solve that you've almost entirely solved the problem yourself.

Gaschnig's algorithm is totally disjoint with AC-3 and forward-checking! They are **not** used in conjunction (it makes no sense to use them in conjunction)!