2.

2. Philosophy is relevant to Teddy's design (and indeed the entire film) primarily by calling into question our ontological assumptions with respect to consciousness and conscious beings. Human-like reactions to his environment such as (seemingly angrily) rejecting being called a toy suggests a sense of self which is arguably the defining feature of conscious life. Current AI systems are often agents who take actions to achieve an objective, with their level of intelligence dictating how effective they are at doing so (but by the orthogonality thesis, having no effect on what the goals are). We are therefore prompted to question, under the assumption that Teddy is highly intelligent, what goal does acting offended maximise? The natural answers are profoundly human instrumental goals, such as being respected, or having his personhood recognised.

3.
- Natural language processing (including realistic speech synthesis)
- Realistic anthropomorphic movement

4.
- See 2.3.
- Reward modelling, possibly through cooperative inverse reinforcement learning. He seems to have an intrinsic understanding that David's "utility function" (whether explicit or implied) is also his own, without having a perfect model of exactly what that function is. An example of this is when David is tricked into cutting Monica's hair, Teddy seems to understand that this will score poorly on their shared utility function, even though David doesn't.

5.

6.
- Ethical reasoning, and loyalty thereto. Teddy seems to have an intrinsic property of "siding with the good character". In the film this is framed as "rejecting his programming". I think this is unlikely to be achieved soon because goal preservation is a convergent instrumental goal. If Teddy's goal was to be a companion to Martin, abandoning Martin and instead wanting to be a companion to David would be a very poor strategy, and so would be unlikely to be chosen by Teddy's AI.

7.

3.

1. If path cost is a non-decreasing function of node depth, then given the set of all paths from node A to node B, the lowest-cost such path (or joint lowest-cost) will be one of those with shortest node depth. Since BFS searches shorter node-depth paths before longer node-depth paths, it is guaranteed to find the optimal path before the non-optimal paths.

2. Let a_n be the number of nodes at depth n.

$a_0 = 1$
$a_{n+1} = ba_n$
$\therefore a_n = b^n$
$f_1(b,d) = \Sigma_{0\leq i<d} a_i$
$= \Sigma_{0\leq i<d} b^i$
$= (1-b^d)/(1-b)$ if $b \mathrel{!}= 1$
d otherwise

$f_2(b,d) = \Sigma_{1\leq i\leq d} f_1(b,i)$

When b is large, $f_1(b,d) \approx b^{d-1}$

$$\begin{aligned} f_2(b,d) &\approx \Sigma_{0\leq i<d} b^i \\ &= (1-b^d)/(1-b) \\ &\approx b^{d-1} \\ &\approx f_1(b,d) \end{aligned}$$

3. This is a misguided idea because while expanding a node (say, *s*), while one of the descendants of *s* may indeed be a goal, there's no guarantee that it's the optimal goal. It may instead be that a sibling of *s* is the optimal goal. It therefore makes sense to push the descendants of *s* to the back of the queue as normal, and check for the goal condition when they are reached.

4. See attachment.

5. These *f* values only need to be stored during the relevant function call (and naturally must be recoverable when its recursive calls return). This means that most of these values can be discarded when they are no longer useful. Therefore, the memory requirement is actually linear with respect to the depth of the tree.

6. This might not be so effective because there is no guarantee that optimal searches combine when added tip-to-tail. For example, consider a start node *A*, an end node *B*, and an additional node, *C*. Suppose that the optimal path from *A* to *B* does not go through *C*. The forward search from *A* might find an optimal path to *C*, at which point it might meet with the backwards search. Taking these two routes together, the algorithm has found the optimal path from *A* to *B* under the restriction that the path must travel through *C*, which the true optimal path does not.

7.

8.
   a. It might be. If the search finds itself at a local maximum < the global maximum, the only way to improve is to temporarily reduce *f*.

   b. Let s = the current node

      $P(\text{move to node } s') \propto (f(s')/f(s))^{\alpha}$

      Where α is a parameter to be tuned. When f(s') > f(s), then f(s')/f(s) is greater than 1, and so exponentiating it makes it even greater. When f(s') < f(s), then f(s')/(f(s) is between 0 and 1, so exponentiating it makes it smaller. Greater values of α correspond to a greater preference for steps which increase f. Probabilities are scaled by a constant such that they sum to 1.

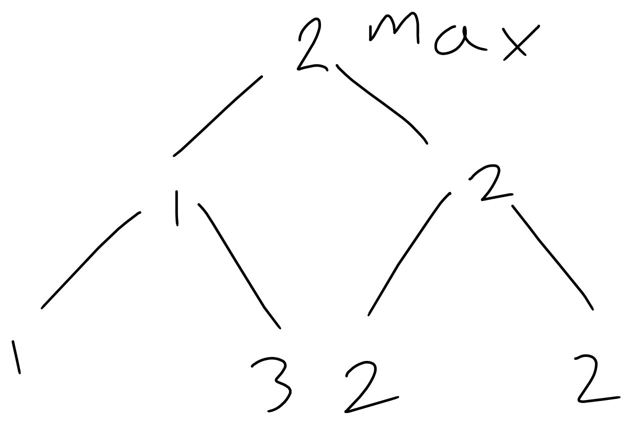
4.
   2.

```
import numpy as np

class Tree:
	def __init__(self, value=None, children=[]):
		self.children = children
		self.value = value
```

```python
    def __str__(self):
        c = list(str(x).split("\n") for x in self.children)
        v = "" if self.value is None else str(self.value)
        n = len(c)

        if n > 3:
            return "I don't know how to print this :("

        if n == 0:
            return v

        tlen = sum(len(x[0]) for x in c) + n - 1

        s = v.center(tlen, " ") + "\n"
        if n == 1:
            s += "|".center(tlen, " ")
        else:
            s += "/".center(len(c[0][0])) + " "
            if n == 3:
                s += "|".center(len(c[1][0])) + " "
            s += "\\".center(len(c[-1][0]))
        s += "\n"

        c = list(list(j for j in x if j.strip() != "") for x in c)
        max_lines = max(len(x) for x in c)
        for i, x in enumerate(c):
            clen = len(x[0])
            while len(c[i]) < max_lines:
                c[i].append(" "*clen)

        for i in range(max_lines):
            s += " ".join(x[i] for x in c) + "\n"

        return s

    @staticmethod
    def from_leaves(branching_factors, leaves):
        assert len(leaves) == np.prod(branching_factors)
        trees = list(map(Tree, leaves))
        for b in branching_factors[::-1]:
            next_layer = []
            while len(trees) > 0:
                children = trees[:b]
                trees = trees[b:]
                next_layer.append(Tree(children=children))
            trees = next_layer
        assert len(trees) == 1
        return trees[0]
```

```
clipped = []
def minimax(root, is_max=True, α=-np.inf, β=np.inf):
        global clipped
        if root.value is not None:
                return root.value
        value = -np.inf if is_max else np.inf
        for i, c in enumerate(root.children):
                cmp = max if is_max else min
                value = cmp(value, minimax(c, not is_max, α, β))
                if is_max:
                        if value >= β:
                                clipped += root.children[i+1:]
                                return value
                        if value < α:
                                α = value
                else:
                        if value <= α:
                                clipped += root.children[i+1:]
                                return value
                        if value < β:
                                β = value
        return value


tree = Tree.from_leaves([3,2,2,2], [1,-15,2,19,18,23,4,3,2,1,7,8,9,10,-
2,5,-1,-30,4,7,20,-1,-1,-5])

print(tree)
print(minimax(tree))
for c in clipped:
        print(c, end="")



##################OUTPUT#################
#
#
#          /                   |                   \
#     /          \        /        \         /           \
#   /    \     /    \   /   \   /    \     /     \    /     \
# /  \  / \  /  \  / \ / \ / \ / \  /  \ /   \  / \ /  \  /  \
# 1 -15 2 19 18 23 4 3 2 1 7 8 9 10 -2 5 -1 -30 4 7 20 -1 -1 -5
#
# 7
#
# / \
# 4 3
#
# /   \
# -2 5
```

3. No. Consider the example:



As shown above, Max (playing minimax) will first choose the right-hand option. However, if Max had chosen the left, Min (an imperfect player) might have chosen 3 instead of 1, which would have been preferable for Max.