Research on human problem-solving suggests that people search through a combinatorial problemspace – a tree where the nodes represent partial solutions, and the branches correspond to operators that modify them [18, 19]. Which branch to take is determined by heuristics that help to navigate the problem-space and guide the problem-solver towards a solution. This perspective highlights two key shortcomings of existing approaches that use LMs to solve general problems: 1) Locally, they do not explore different continuations within a thought process – the branches of the tree. 2) Globally, they do not incorporate any type of planning, lookahead, or backtracking to help evaluate these different options – the kind of heuristic-guided search that seems characteristic of human problem-solving. To address these shortcomings, we introduce Tree of Thoughts (ToT), a paradigm that allows LMs to explore multiple reasoning paths over thoughts (Figure 1(c)). ToT frames any problem as a search over a tree, where each node is a state s = [x, z1···i ] representing a partial solution with the input and the sequence of thoughts so far. A specific instantiation of ToT involves answering four questions: 1. How to decompose the intermediate process into thought steps; 2. How to generate potential thoughts from each state; 3. How to heuristically evaluate states; 4. What search algorithm to use. 1. Thought decomposition. While CoT samples thoughts coherently without explicit decomposition, ToT leverages problem properties to design and decompose intermediate thought steps. As Table 1 shows, depending on different problems, a thought could be a couple of words (Crosswords), a line of equation (Game of 24), or a whole paragraph of writing plan (Creative Writing). In general, a thought should be “small” enough so that LMs can generate promising and diverse samples (e.g. generating a whole book is usually too “big” to be coherent), yet “big” enough so that LMs can evaluate its prospect toward problem solving (e.g. generating one token is usually too “small” to evaluate). 2. Thought generator G(pθ, s, k). Given a tree state s = [x, z1···i ], we consider two strategies to generate k candidates for the next thought step: (a) Sample i.i.d. thoughts from a CoT prompt (Creative Writing, Figure 4): z (j) ∼ p CoT θ (zi+1|s) = p CoT θ (zi+1|x, z1···i) (j = 1 · · · k). This works better when the thought space is rich (e.g. each thought is a paragraph), and i.i.d. samples lead to diversity; (b) Propose thoughts sequentially using a “propose prompt” (Game of 24, Figure 2; Crosswords, Figure 6): [z (1) , · · · , z(k) ] ∼ p propose θ (z (1···k) i+1 | s). This works better when the thought space is more constrained (e.g. each thought is just a word or a line), so proposing different thoughts in the same context avoids duplication. 3. State evaluator V (pθ, S). Given a frontier of different states, the state evaluator evaluates the progress they make towards solving the problem, serving as a heuristic for the search algorithm to determine which states to keep exploring and in which order. While heuristics are a standard approach to solving search problems, they are typically either programmed (e.g. DeepBlue [3]) or 3 learned (e.g. AlphaGo [26]). We propose a third alternative, by using the LM to deliberately reason about states. When applicable, such a deliberate heuristic can be more flexible than programmed rules, and more sample-efficient than learned models. Similar to the thought generator, we consider two strategies to evaluate states either independently or together: (a) Value each state independently: V (pθ, S)(s) ∼ p value θ (v|s) ∀s ∈ S, where a value prompt reasons about the state s to generate a scalar value v (e.g. 1-10) or a classification (e.g. sure/likely/impossible) that could be heuristically turned into a value. The basis of such evaluative reasoning can vary across problems and thought steps. In this work, we explore evaluation via few lookahead simulations (e.g. quickly confirm that 5, 5, 14 can reach 24 via 5 + 5 + 14, or “hot l” can mean “inn” via filling “e” in “ ”) plus commonsense (e.g. 1 2 3 are too small to reach 24, or no word can start with “tzxc”). While the former might promote “good” states, the latter could help eliminate “bad” states. Such valuations do not need to be perfect, and only need to be approximately (b) Vote across states: V (pθ, S)(s) = 1[s = s ∗ ], where a “good” state s ∗ ∼ p vote θ (s ∗ |S) is voted out based on deliberately comparing different states in S in a vote prompt. When problem success is harder to directly value (e.g. passage coherency), it is natural to to instead compare different partial solutions and vote for the most promising one. This is similar in spirit to a “step-wise” self-consistency strategy, i.e. cast “which state to explore” as a multi-choice QA, and use LM samples to vote for it. For both strategies, we could prompt the LM multiple times to aggregate the value or vote results to trade time/resource/cost for more faithful/robust heuristics.

Algorithm 1 ToT-BFS(x, pθ, G, k, V, T, b) Require: Input x, LM pθ, thought generator G() & size limit k, states evaluator V (), step limit T, breadth limit b. S0 ← {x} for t = 1, · · · , T do S 0 t ← {[s, z] | s ∈ St−1, zt ∈ G(pθ, s, k)} Vt ← V (pθ, S0 t ) St ← arg maxS⊂S0 t ,|S|=b P s∈S Vt(s) end for return G(pθ, arg maxs∈ST VT (s), 1)

Algorithm 2 ToT-DFS(s, t, pθ, G, k, V, T, vth) Require: Current state s, step t, LM pθ, thought generator G() and size limit k, states evaluator V (), step limit T, threshold vth if t > T then record output G(pθ, s, 1) end if for s 0 ∈ G(pθ, s, k) do . sorted candidates if V (pθ, {s 0})(s) > vthres then . pruning DFS(s 0 , t + 1) end if end for

Search algorithm.

Finally, within the ToT framework, one can plug and play different search algorithms depending on the tree structure. We explore two relatively simple search algorithms and leave more advanced ones (e.g. A\* [9], MCTS [2]) for future work: (a) Breadth-first search (BFS) (Algorithm 1) maintains a set of the b most promising states per step. This is used for Game of 24 and Creative Writing where the tree depth is limit (T ≤ 3), and initial thought steps can be evaluated and pruned to a small set (b ≤ 5). (b) Depth-first search (DFS) (Algorithm 2) explores the most promising state first, until the final output is reached (t > T), or the state evaluator deems it impossible to solve the problem from the current s (V (pθ, {s})(s) ≤ vth for a value threshold vth). In the latter case, the subtree from s is pruned to trade exploration for exploitation. In both cases, DFS backtracks to the parent state of s to continue exploration. Conceptually, ToT has several benefits as a method for general problem-solving with LMs: (1) Generality. IO, CoT, CoT-SC, and self-refinement can be seen as special cases of ToT (i.e. trees of limited depth and breadth; Figure 1). (2) Modularity. The base LM, as well as the thought decomposition, generation, evaluation, and search procedures can all be varied independently. (3) Adaptability. Different problem properties, LM capabilities, and resource constraints can be accommodated. (4) Convenience. No extra training is needed, just a pre-trained LM is sufficient.

def ToT\_BFS(x, p\_theta, thought\_generator, k, state\_evaluator, T, b):

S0 = {x}

for t in range(1, T + 1):

S0\_t = set([(s, []) for s in S0])

V\_t = state\_evaluator(p\_theta, S0\_t)

St = max(S0\_t, key=lambda s: V\_t[s])

if t == T:

return thought\_generator(p\_theta, St[0], 1, b)

S\_t = set()

for s in St:

thoughts = thought\_generator(p\_theta, s[0], k, b)

for thought in thoughts:

if state\_evaluator(p\_theta, (thought, s[1])) > 0:

S\_t.add((thought, s[1] + [thought]))

S0 = S\_t

return None

def thought\_generator(p\_theta, s, k, b):

# Implement the thought generator function specific to Crystal Ball

# Generate thoughts based on the current state s, parameter p\_theta, k, and b

# Return a list of generated thoughts

thoughts = []

# Your implementation here

return thoughts

def state\_evaluator(p\_theta, s):

# Implement the state evaluator function specific to Crystal Ball

# Evaluate the state s based on the parameter p\_theta

# Return the evaluation score

score = 0

# Your implementation here

return score

# Example usage:

x = ...

p\_theta = ...

k = ...

T = ...

b = ...

result = ToT\_BFS(x, p\_theta, thought\_generator, k, state\_evaluator, T, b)