Greedy Algorithms for Cardinality-Constrained Submodular Maximization

Feier Lyu

June 29, 2026

Abstract

We formalize in Isabelle/HOL the classical approximation guarantee for monotone non-negative submodular maximization under a cardinality constraint on a finite ground set. The main result is the Nemhauser–Wolsey bound for deterministic greedy: after k steps, the greedy solution achieves the finite-step guarantee $1 - (1 - 1/k)^k$, and hence also the standard $(1 - 1/e)$ approximation ratio. The development also includes a verified stateful lazy greedy variant, based on cached upper bounds on marginal gains, and proves that it satisfies the same approximation guarantee.

Contents

Background

The classical greedy algorithm for monotone submodular maximization under a cardinality constraint is due to Nemhauser, Wolsey, and Fisher [2]. This entry also formalizes a stateful lazy-greedy variant, following the accelerated greedy idea of Minoux [1]: instead of recomputing all marginal gains at every outer iteration, the algorithm keeps cached upper bounds on marginal gains and recomputes them only when needed. The formalization proves that this lazy implementation still selects a valid greedy element at each step and therefore inherits the same approximation guarantee.

```
theory Submodular_Base
  imports Complex_Main
begin

lemma finite_has_maximal_on:
  fixes g :: "'a ⇒ 'b::linorder"
  assumes fin: "finite A"
    and nonempty: "A ≠ {}"
  shows "∃x∈A. ∀y∈A. g y ≤ g x" using arg_max_on_def[of g A]
proof -
  have fin_image: "finite (g ` A)"
    using fin by simp
  have nonempty_image: "g ` A ≠ {}"
    using nonempty by auto

  have max_in_image: "Max (g ` A) ∈ g ` A"
    using Max_in[OF fin_image nonempty_image] .

  then obtain x where xA: "x ∈ A" and x_eq: "g x = Max (g ` A)"
    by auto

  have "∀y∈A. g y ≤ g x"
    by (simp add: fin_image x_eq)

  then show ?thesis
    using xA by blast
```

qed

The main development is carried out in a single locale for normalized monotone submodular functions, which is the setting needed for the cardinality-constrained greedy approximation guarantees below. Some auxiliary facts hold under weaker assumptions; for this entry we keep them in the same locale to maintain a compact and uniform development.

```
locale Submodular_Func =
  fixes V :: "'a set" and f :: "'a set ⇒ real"
  assumes finite_V: "finite V"
      and monotone_f: "⋀S T. S ⊆ T ⟹ T ⊆ V ⟹ f S ≤ f T"
      and submodular_f:
        "⋀S T. S ⊆ V ⟹ T ⊆ V ⟹ f (S ∪ T) + f (S ∩ T) ≤ f S + f
T"
      and f_empty: "f {} = 0"
begin
```

Marginal gain of adding a single element to a set.

```
definition gain :: "'a set ⇒ 'a ⇒ real" where
  "gain S e = f (S ∪ {e}) - f S"

lemma f_nonneg:
  assumes "S ⊆ V"
  shows "0 ≤ f S"
proof -
  have "{} ⊆ S" by auto
  from monotone_f[OF this assms] have "f {} ≤ f S" .
  thus ?thesis by (simp add: f_empty)
qed

lemma monotone_on_PowV:
  shows "monotone_on (Pow V) (⊆) (≤) f"
  unfolding monotone_on_def
  using monotone_f by auto

lemma gain_nonneg:
  assumes "S ⊆ V" and "x ∈ V - S"
  shows "0 ≤ gain S x"
proof -
  have "S ⊆ S ∪ {x}" by auto
  moreover from assms have "S ∪ {x} ⊆ V" by auto
  ultimately have "f S ≤ f (S ∪ {x})" using monotone_f by auto
  thus ?thesis by (simp add: gain_def)
qed
```

Diminishing returns for single-element marginal gains.

```
lemma gain_decreasing:
  assumes "S ⊆ T" "T ⊆ V" "x ∈ V" "x ∉ T"
  shows "gain S x ≥ gain T x"
```

```
proof -
  have Sx_sub_V: "insert x S ⊆ V"
    using assms by auto

  have subm:
    "f (insert x (S ∪ T)) + f (insert x S ∩ T) ≤ f (insert x S) + f T"
    using submodular_f[OF Sx_sub_V assms(2)]
    by simp

  have inter_eq: "insert x S ∩ T = S"
    using assms by auto

  have union_eq: "insert x (S ∪ T) = insert x T"
    using assms by auto

  from subm have "f S + f (insert x T) ≤ f T + f (insert x S)"
    by (simp add: inter_eq union_eq)

  hence "f (insert x S) - f S ≥ f (insert x T) - f T"
    by linarith

  thus ?thesis
    by (simp add: gain_def)
qed
```

Set-valued diminishing returns.

```
lemma gain_decreasing_set:
  assumes "S ⊆ T" "T ⊆ V" "A ⊆ V"
  shows "f (S ∪ A) - f S ≥ f (T ∪ A) - f T"
proof -
  have SUA_subV: "S ∪ A ⊆ V"
    using assms by auto

  have subm:
    "f ((S ∪ A) ∪ T) + f ((S ∪ A) ∩ T) ≤ f (S ∪ A) + f T"
    using submodular_f[OF SUA_subV assms(2)] .

  have union_eq: "(S ∪ A) ∪ T = T ∪ A"
    using assms by auto

  have S_sub_inter: "S ⊆ (S ∪ A) ∩ T"
    using assms by auto

  have inter_subV: "(S ∪ A) ∩ T ⊆ V"
    using assms by auto

  have mono_inter: "f S ≤ f ((S ∪ A) ∩ T)"
    using monotone_f[OF S_sub_inter inter_subV] .
```

```
  have "f (T ∪ A) + f ((S ∪ A) ∩ T) ≤ f (S ∪ A) + f T"
    using subm by (simp add: union_eq)
  then have "f (T ∪ A) + f S ≤ f (S ∪ A) + f T"
    using mono_inter by linarith
  then show ?thesis
    by linarith
qed

end
```

This entry treats cardinality-constrained monotone submodular maximization. Other constraint systems, such as matroid or knapsack constraints, are outside the scope of the present development.

```
locale Cardinality_Constraint = Submodular_Func +
  fixes k :: nat
  assumes k_le_cardV: "k ≤ card V"
begin

definition feasible :: "'a set ⇒ bool" where
  "feasible S ⟷ S ⊆ V ∧ card S ≤ k"

lemma feasibleI:
  assumes "S ⊆ V" "card S ≤ k"
  shows "feasible S"
  using assms unfolding feasible_def by auto

lemma feasibleD:
  assumes "feasible S"
  shows "S ⊆ V" "card S ≤ k"
  using assms unfolding feasible_def by auto

lemma feasible_empty[simp]: "feasible {}"
  unfolding feasible_def by auto

lemma feasible_family_nonempty: "Collect feasible ≠ {}"
proof -
  have "∃S. feasible S"
  proof
    show "feasible {}" by (rule feasible_empty)
  qed
  then show ?thesis by auto
qed

lemma finite_feasible_family: "finite {S. feasible S}"
proof -
  have "{S. feasible S} ⊆ Pow V"
    by (auto simp: feasible_def)
  moreover have "finite (Pow V)"
    using finite_V by simp
```

```
  ultimately show ?thesis
    by (rule finite_subset)
qed
```

0.1 Optimal feasible sets

We select a canonical optimal feasible set *OPT_set* using Hilbert choice and define *OPT_k* as its value. These are problem-level objects for the cardinality-constrained maximization problem, independent of any particular greedy implementation.

```
definition OPT_set :: "'a set" where
  "OPT_set =
     (SOME X. feasible X ∧ (∀Y. feasible Y ⟶ f Y ≤ f X))"

lemma exists_max_feasible:
  "∃X. feasible X ∧ (∀Y. feasible Y ⟶ f Y ≤ f X)"
proof -
  from finite_has_maximal_on[OF finite_feasible_family feasible_family_nonempty]
  obtain X where X_feas: "X ∈ Collect feasible"
    and X_max: "∀Y ∈ Collect feasible. f Y ≤ f X"
    by blast
  have "feasible X ∧ (∀Y. feasible Y ⟶ f Y ≤ f X)"
    using X_feas X_max by auto
  thus ?thesis ..
qed

lemma OPT_set_props:
  shows OPT_set_in: "feasible OPT_set"
    and OPT_set_max: "∀Y. feasible Y ⟶ f Y ≤ f OPT_set"
proof -
  from exists_max_feasible
  obtain X where X_in: "feasible X"
    and X_max: "∀Y. feasible Y ⟶ f Y ≤ f X"
    by blast
  then have ex_spec:
    "∃X. feasible X ∧ (∀Y. feasible Y ⟶ f Y ≤ f X)"
    by blast
  from someI_ex[OF ex_spec]
  have "feasible OPT_set ∧ (∀Y. feasible Y ⟶ f Y ≤ f OPT_set)"
    unfolding OPT_set_def by simp
  then show "feasible OPT_set"
    and "∀Y. feasible Y ⟶ f Y ≤ f OPT_set"
    by auto
qed

definition OPT_k :: real where
  "OPT_k = f OPT_set"

lemma exists_opt_set:
```

```
  "∃X. feasible X ∧ f X = OPT_k"
proof -
  have "feasible OPT_set"
    by (rule OPT_set_in)
  moreover have "f OPT_set = OPT_k"
    unfolding OPT_k_def by simp
  ultimately show ?thesis
    by blast
qed

lemma OPT_k_upper_bound:
  assumes "feasible S"
  shows "f S ≤ OPT_k"
proof -
  have "∀Y. feasible Y ⟶ f Y ≤ f OPT_set"
    by (rule OPT_set_max)
  with assms have "f S ≤ f OPT_set"
    by auto
  thus ?thesis
    unfolding OPT_k_def by simp
qed
```

0.2 Basic non-emptiness facts

```
lemma nonempty_candidates:
  assumes "S ⊆ V" "card S < k"
  shows "V - S ≠ {}"
proof
  assume "V - S = {}"
  hence "V ⊆ S" by auto
  with assms(1) have "V = S" by auto
  with assms(2) k_le_cardV show False by simp
qed

lemma nonempty_gap:
  assumes "S ⊆ V" "Opt ⊆ V" "f S < f Opt"
  shows "Opt - S ≠ {}"
proof
  assume "Opt - S = {}"
  hence "Opt ⊆ S" by auto
  with assms(1,2) have "f Opt ≤ f S"
    using monotone_f by auto
  with assms(3) show False by linarith
qed

lemma OPT_k_nonneg: "0 ≤ OPT_k"
proof -
  have "feasible {}"
    by (rule feasible_empty)
```

```
  then have "f {} ≤ OPT_k"
    by (rule OPT_k_upper_bound)
  thus ?thesis
    by (simp add: f_empty)
qed
```

Submodular telescoping: sum of marginals upper-bounds the joint gain.

```
lemma submod_sum_upper:
  assumes "finite A" "A ⊆ V" "S ⊆ V" "A ∩ S = {}"
  shows "f (S ∪ A) - f S ≤ (∑x∈A. gain S x)"
  using assms
proof (induction rule: finite_induct)
  case empty
  then show ?case by simp
next
  case (insert a A)
  from insert.hyps have a_notin: "a ∉ A" and finA: "finite A" by auto

  from insert.prems have S_sub: "S ⊆ V" by auto
  from insert.prems have ins_subV: "insert a A ⊆ V" by auto
  from insert.prems have ins_disj: "insert a A ∩ S = {}" by auto

  have A_sub: "A ⊆ V" using ins_subV by auto
  have aV   : "a ∈ V"  using ins_subV by auto
  have disjA: "A ∩ S = {}" using ins_disj by auto
  have a_notS: "a ∉ S" using ins_disj by auto

  have step:
    "f (S ∪ insert a A) - f S
     = (f ((S ∪ A) ∪ {a}) - f (S ∪ A)) + (f (S ∪ A) - f S)"
    by (simp add: insert_commute Un_assoc)

  have SSUA: "S ⊆ S ∪ A" by auto
  have SUA_subV: "S ∪ A ⊆ V" using S_sub A_sub by auto
  have a_notin_SUA: "a ∉ S ∪ A" using a_notS a_notin by auto
  have dec:
    "f ((S ∪ A) ∪ {a}) - f (S ∪ A) ≤ gain S a"
    using gain_decreasing[OF SSUA SUA_subV aV a_notin_SUA]
    by (simp add: gain_def)

  from insert.IH[OF A_sub S_sub disjA]
  have IH: "f (S ∪ A) - f S ≤ (∑x∈A. gain S x)" .

  have "(f ((S ∪ A) ∪ {a}) - f (S ∪ A)) + (f (S ∪ A) - f S)
        ≤ gain S a + (∑x∈A. gain S x)"
    using dec IH by linarith
  thus ?case
    by (simp add: step insert_commute finA a_notin)
qed
```

Average marginal bound against any candidate set *Opt* $\subseteq$ *V* with *card Opt* $\leq$ *k*: if *S* $\subseteq$ *V* and *card S* < *k*, then there exists an element *e* $\in$ *V - S* such that *gain S e* $\geq$ *(f Opt - f S) / real k*.

```
lemma marginal_gain_lower_bound:
  fixes Opt S :: "'a set"
  assumes S_sub: "S ⊆ V"
    and O_sub: "Opt ⊆ V"
    and cardS_lt_k: "card S < k"
    and cardO_le_k: "card Opt ≤ k"
  shows "∃e∈V - S. gain S e ≥ (f Opt - f S) / real k"
proof -
  have finV: "finite V" by (rule finite_V)
  have k_pos: "0 < k" using cardS_lt_k by (simp add: not_less)

  consider (le) "f Opt ≤ f S" | (gt) "f S < f Opt" by linarith
  then show ?thesis
  proof cases
    case le
    have VS_ne: "V - S ≠ {}"
      using nonempty_candidates[OF S_sub cardS_lt_k] .

    then obtain e where eVS: "e ∈ V - S" by blast
    hence ge0: "0 ≤ gain S e" using S_sub gain_nonneg by auto

    moreover have "(f Opt - f S) / real k ≤ 0"
    proof -
      have "f Opt - f S ≤ 0" using le by linarith
      thus ?thesis
        using k_pos by (simp add: divide_nonpos_pos)
    qed

    ultimately have "(f Opt - f S) / real k ≤ gain S e"
      by linarith

    thus ?thesis
      using eVS by (intro bexI[of _ e]) auto
  next
    case gt
    have OS_ne: "Opt - S ≠ {}"
      using nonempty_gap[OF S_sub O_sub gt] .

    have finOS: "finite (Opt - S)"
      using finV O_sub by (meson Diff_subset finite_subset)
    have OS_subV: "Opt - S ⊆ V" using O_sub by auto
    have disj: "(Opt - S) ∩ S = {}" by auto
    have finO: "finite Opt" using finV O_sub finite_subset by blast

    have step_sum:
      "f (S ∪ (Opt - S)) - f S ≤ (∑x∈Opt - S. gain S x)"
```

```
  using submod_sum_upper[OF finOS OS_subV S_sub disj] .

have SUO_subV: "S ∪ Opt ⊆ V" using S_sub O_sub by auto
have sum_upper: "f Opt - f S ≤ (∑x∈Opt - S. gain S x)"
proof -
  have "f Opt ≤ f (S ∪ Opt)"
    using monotone_f[rule_format, of Opt "S ∪ Opt"] SUO_subV by auto
  then have "f Opt - f S ≤ f (S ∪ Opt) - f S" by linarith
  also have "S ∪ Opt = S ∪ (Opt - S)" by auto
  also have "f (S ∪ (Opt - S)) - f S
             ≤ (∑x∈Opt - S. gain S x)"
    using step_sum .
  finally show ?thesis .
qed

obtain e where e_in: "e ∈ Opt - S"
  and e_max: "∀y∈Opt - S. gain S y ≤ gain S e"
  using finite_has_maximal_on[OF finOS OS_ne, of "gain S"]
  by blast

have "(∑x∈Opt - S. gain S x) ≤ (∑x∈Opt - S. gain S e)"
  using e_max by (intro sum_mono) simp_all
also have "... = real (card (Opt - S)) * gain S e"
  by simp
finally have sum_le_card_max:
  "(∑x∈Opt - S. gain S x) ≤ real (card (Opt - S)) * gain S e" .

have base: "f Opt - f S ≤ real (card (Opt - S)) * gain S e"
  using sum_upper sum_le_card_max by linarith

have cardOS_le_k: "card (Opt - S) ≤ k"
proof -
  have "card (Opt - S) ≤ card Opt"
    using finO Diff_subset by (rule card_mono)
  also have "... ≤ k" using cardO_le_k .
  finally show ?thesis .
qed

have eVS: "e ∈ V - S" using e_in O_sub by auto
have ge0: "0 ≤ gain S e" using S_sub eVS gain_nonneg by auto

have "real (card (Opt - S)) * gain S e ≤ real k * gain S e"
  using cardOS_le_k ge0 by (simp add: mult_right_mono)
hence main_ineq: "f Opt - f S ≤ real k * gain S e"
  using base by linarith

have "gain S e ≥ (f Opt - f S) / real k"
  using main_ineq k_pos by (simp add: mult.commute pos_divide_le_eq)
thus ?thesis using eVS by blast
```

```
  qed
qed

end

end
theory Greedy_Submodular_Construct
  imports "../Core/Submodular_Base"
begin
```

1 Greedy construction

This theory sets up the greedy construction for monotone submodular maximization under a cardinality constraint. The locale `Greedy_Setup` fixes a finite ground set `V`, a budget `k`, and a normalized monotone submodular set function `f`. The greedy sequence starts from the empty set and repeatedly adds an element of maximum marginal gain, with ties broken by the abstract oracle.

1.1 Preliminaries on finite maximizers

Finite arg-max via the standard maximality predicate.

```
lemma finite_is_arg_max_in:
  fixes g :: "'a ⇒ 'b::linorder"
  assumes fin: "finite A" and ne: "A ≠ {}"
  shows "∃x∈A. is_arg_max g (λx. x ∈ A) x"
proof -
  have img_fin: "finite (g ` A)"
    using fin by simp
  have img_ne: "g ` A ≠ {}"
    using ne by auto

  let ?M = "Max (g ` A)"
  have M_in: "?M ∈ g ` A"
    using Max_in[OF img_fin img_ne] .
  then obtain x where xA: "x ∈ A" and gx: "g x = ?M"
    by auto

    have no_better: "¬ (∃y. y ∈ A ∧ g y > g x)"
    proof
      assume ex: "∃y. y ∈ A ∧ g y > g x"
      then obtain y where yA: "y ∈ A" and gy: "g y > g x" by auto

      have "g y ≤ Max (g ` A)"
        using Max_ge_iff[OF img_fin img_ne] yA by auto
      hence gy_le_gx: "g y ≤ g x"
        by (simp add: gx)
```

```
      have gx_lt_gy: "g x < g y" using gy by simp
      show False
        using gx_lt_gy gy_le_gx by (meson less_le_not_le)
    qed

  have "is_arg_max g (λz. z ∈ A) x"
    unfolding is_arg_max_def
    using xA no_better by auto
  thus ?thesis
    using xA by blast
qed

lemma is_arg_maxD_le:
  fixes f :: "'b ⇒ 'a::linorder"
  assumes "is_arg_max f P x" "P y"
  shows "f y ≤ f x"
  using assms
  by (simp add: is_arg_max_linorder)
```

Abstract setup for the greedy algorithm: a finite ground set V, a budget k, and a non-negative monotone submodular function f with f = 0.

This theory focuses on the greedy construction and basic structural properties, without yet proving approximation guarantees.

1.2 Hilbert-choice arg-max oracle for marginal gain

```
context Submodular_Func
begin

definition argmax_gain_some :: "'a set ⇒ 'a set ⇒ 'a" where
  "argmax_gain_some S A =
     (SOME x. x ∈ A ∧ is_arg_max (gain S) (λy. y ∈ A) x)"

lemma argmax_gain_some_mem:
  assumes fin: "finite A" and ne: "A ≠ {}"
  shows "argmax_gain_some S A ∈ A"
proof -
  have exB: "∃x∈A. is_arg_max (gain S) (λy. y ∈ A) x"
    using finite_is_arg_max_in[OF fin ne] .

  have ex: "∃x. x ∈ A ∧ is_arg_max (gain S) (λy. y ∈ A) x"
    using exB by auto

  show ?thesis
    unfolding argmax_gain_some_def
    using someI_ex[OF ex] by blast
qed
```

```
lemma argmax_gain_some_max:
  assumes fin: "finite A" and ne: "A ≠ {}" and yA: "y ∈ A"
  shows "gain S y ≤ gain S (argmax_gain_some S A)"
proof -
  have exB: "∃x∈A. is_arg_max (gain S) (λz. z ∈ A) x"
    using finite_is_arg_max_in[OF fin ne] .

  have ex: "∃x. x ∈ A ∧ is_arg_max (gain S) (λz. z ∈ A) x"
    using exB by auto

  have chosen:
    "is_arg_max (gain S) (λz. z ∈ A) (argmax_gain_some S A)"
    unfolding argmax_gain_some_def
    using someI_ex[OF ex] by blast

  show ?thesis
    using is_arg_maxD_le[OF chosen yA] .
qed

end

locale Greedy_Setup = Cardinality_Constraint V f k
  for V :: "'a set" and f :: "'a set ⇒ real" and k :: nat +
  fixes argmax_gain :: "'a set ⇒ 'a set ⇒ 'a"
  assumes argmax_gain_mem:
    "finite A ⟹ A ≠ {} ⟹ argmax_gain S A ∈ A"
  assumes argmax_gain_max:
    "finite A ⟹ A ≠ {} ⟹ ∀y∈A. gain S y ≤ gain S (argmax_gain S
A)"
begin
```

Greedy construction: start from `{}` and, as long as there are remaining elements, add one with maximum marginal gain.

```
fun greedy_set :: "nat ⇒ 'a set" where
  "greedy_set 0 = {}"
| "greedy_set (Suc i) =
     (let S = greedy_set i in
      if V - S = {} then S else insert (argmax_gain S (V - S)) S)"
```

Greedy sets are always subsets of the ground set `V`.

```
lemma greedy_subset_V: "greedy_set i ⊆ V"
proof (induction i)
  case 0
  show ?case by simp
next
  case (Suc i)
  have IH: "greedy_set i ⊆ V" by fact
  show ?case
  proof (cases "V - greedy_set i = {}")
```

```
    case True
    have "greedy_set (Suc i) = greedy_set i"
      using True by (simp add: Let_def)
    thus ?thesis
      using IH by simp

  next
    case False
    have finA: "finite (V - greedy_set i)"
      using finite_V by simp
    have inA:
      "argmax_gain (greedy_set i) (V - greedy_set i) ∈ V - greedy_set
i"
      using argmax_gain_mem[OF finA False] .

    hence inV: "argmax_gain (greedy_set i) (V - greedy_set i) ∈ V"
      by simp

    from IH inV
    have "greedy_set i ∪ {argmax_gain (greedy_set i) (V - greedy_set
i)} ⊆ V"
      by auto
    with False show ?thesis
      by (simp add: Let_def)
  qed
qed
```

If a genuinely new element is inserted, the cardinality increases by one.

```
lemma greedy_card_step:
  assumes ne: "V - greedy_set i ≠ {}"
  shows   "card (greedy_set (Suc i)) = card (greedy_set i) + 1"
proof -
  let ?S = "greedy_set i"
  let ?A = "V - ?S"
  let ?x = "argmax_gain ?S ?A"

  have finS: "finite ?S"
    using greedy_subset_V finite_V
    by (meson finite_subset)

  have finA: "finite ?A"
    using finite_V by simp

  have x_in_A: "?x ∈ ?A"
    using argmax_gain_mem[OF finA ne] .
  hence x_notin_S: "?x ∉ ?S"
    by simp

  have suc_def:
```

```
    "greedy_set (Suc i) =
       (let S = ?S in
        if V ⊆ S then S else insert (argmax_gain S (V - S)) S)"
    by simp

  from ne have "greedy_set (Suc i) = ?S ∪ {?x}"
    unfolding suc_def by (simp add: Let_def)
  hence "greedy_set (Suc i) = insert ?x ?S"
    by simp

  thus ?thesis
    using finS x_notin_S by simp
qed
```

If the remainder is empty, the greedy set stays unchanged.

```
lemma greedy_card_idle:
  assumes "V - greedy_set i = {}"
  shows   "card (greedy_set (Suc i)) = card (greedy_set i)"
  using assms by (simp add: Let_def)
```

State transition: when the remainder is non-empty, `S` evolves by adding the arg-max element.

```
lemma state_transition_nonempty:
  assumes "0 < i" and "V - greedy_set (i - 1) ≠ {}"
  shows   "greedy_set i =
           greedy_set (i - 1)
           ∪ {argmax_gain (greedy_set (i - 1)) (V - greedy_set (i - 1))}"
proof -
  obtain j where ij: "i = Suc j"
    using assms(1) by (cases i) auto

  from assms(2) ij have rem_ne: "V - greedy_set j ≠ {}"
    by simp

  have def_suc:
    "greedy_set (Suc j) =
       (let S = greedy_set j in
        if V ⊆ S then S else insert (argmax_gain S (V - S)) S)"
    by simp

  from rem_ne have
    "greedy_set (Suc j) =
       greedy_set j ∪ {argmax_gain (greedy_set j) (V - greedy_set j)}"
    by (simp add: def_suc Let_def)

  with ij show ?thesis
    by simp
qed
```

At most one new element is added in each greedy step.

```
lemma card_greedy_le_i: "card (greedy_set i) ≤ i"
proof (induction i)
  case 0
  show ?case by simp
next
  case (Suc i)
  show ?case
  proof (cases "V - greedy_set i = {}")
    case True
    have "card (greedy_set (Suc i)) = card (greedy_set i)"
      using True by (rule greedy_card_idle)
    with Suc.IH show ?thesis by simp
  next
    case False
    have "card (greedy_set (Suc i)) = card (greedy_set i) + 1"
      using False by (rule greedy_card_step)
    with Suc.IH show ?thesis by simp
  qed
qed
```

If `i ≤ k` then `card (greedy_set i) ≤ k` (used later in the gap bound).

```
lemma card_greedy_le_k:
  assumes "i ≤ k"
  shows   "card (greedy_set i) ≤ k"
  using card_greedy_le_i assms by (meson le_trans)
```

If `card (greedy_set t) < card V`, then the remainder `V - greedy_set t` is non-empty.

```
lemma remainder_nonempty_if_card_ltV:
  assumes "card (greedy_set t) < card V"
  shows   "V - greedy_set t ≠ {}"
proof
  assume "V - greedy_set t = {}"
  then have vsub: "V ⊆ greedy_set t" by auto
  have ssub: "greedy_set t ⊆ V" by (rule greedy_subset_V)
  from ssub vsub have "greedy_set t = V" by (rule subset_antisym)
  with assms show False by simp
qed
```

Under `k ≤ card V`, the greedy transition up to step `k` always adds a new element.

```
lemma state_transition_upto_k:
  assumes "0 < i" "i ≤ k"
  shows   "greedy_set i =
           greedy_set (i - 1)
           ∪ {argmax_gain (greedy_set (i - 1)) (V - greedy_set (i - 1))}"
proof -
  have "card (greedy_set (i - 1)) ≤ i - 1"
    using card_greedy_le_i by simp
```

```
  also have "... < k"
    using assms(1,2) by simp
  also have "... ≤ card V"
    using k_le_cardV by simp
  finally have ltV: "card (greedy_set (i - 1)) < card V" .

  have rem_ne: "V - greedy_set (i - 1) ≠ {}"
    using remainder_nonempty_if_card_ltV[OF ltV] .

  show ?thesis
    by (rule state_transition_nonempty[OF assms(1) rem_ne])
qed
```

Intermediate greedy states as a list $[S\,0, \ldots, S\,n]$.

```
definition greedy_sequence :: "nat ⇒ 'a set list" where
  "greedy_sequence n = map greedy_set [0..<Suc n]"
```

Indexing lemma for the sequence representation.

```
lemma greedy_sequence_nth[simp]:
  assumes "i ≤ n"
  shows   "greedy_sequence n ! i = greedy_set i"
proof -
  have i_lt: "i < Suc n" using assms by simp
  have "(map greedy_set [0..<Suc n]) ! i = greedy_set ([0..<Suc n] ! i)"
    using i_lt by (simp only: nth_map length_upt)
  also have "... = greedy_set i"
    using i_lt by (simp only: nth_upt add_0_left)
  finally show ?thesis
    by (simp only: greedy_sequence_def)
qed
```

Every greedy state is finite.

```
lemma greedy_set_finite [simp]: "finite (greedy_set i)"
  using greedy_subset_V finite_V by (meson finite_subset)
```

One-step monotonicity: $S\,i \subseteq S\,(\mathrm{Suc}\ i)$.

```
lemma greedy_mono_Suc: "greedy_set i ⊆ greedy_set (Suc i)"
proof (cases "V - greedy_set i = {}")
  case True
  then show ?thesis by (simp add: Let_def)
next
  case False
  then show ?thesis by (auto simp: Let_def)
qed
```

Chain monotonicity: if $i \le j$ then $S\,i \subseteq S\,j$.

```
lemma greedy_chain_mono:
  assumes "i ≤ j"
  shows   "greedy_set i ⊆ greedy_set j"
```

```
using assms
proof (induction j arbitrary: i)
  case 0
  then show ?case
    by simp
next
  case (Suc j i)
  show ?case
  proof (cases "i ≤ j")
    case True
    with Suc.IH have "greedy_set i ⊆ greedy_set j" .
    also have "... ⊆ greedy_set (Suc j)"
      by (rule greedy_mono_Suc)
    finally show ?thesis .
  next
    case False
    hence "i = Suc j" using Suc.prems by simp
    thus ?thesis by simp
  qed
qed
```

Cardinality is non-decreasing along the greedy sequence.

```
lemma greedy_card_mono:
  "i ≤ j ⟹ card (greedy_set i) ≤ card (greedy_set j)"
  by (meson greedy_chain_mono greedy_set_finite finite_subset card_mono)
```

A compact cardinality bound: `card S ≤ min i (card V)` for all `i`.

```
lemma greedy_card_min:
  "card (greedy_set i) ≤ min i (card V)"
proof -
  have A1: "card (greedy_set i) ≤ i"
    by (rule card_greedy_le_i)
  have A2: "card (greedy_set i) ≤ card V"
  proof -
    have fin: "finite V" using finite_V .
    have sub: "greedy_set i ⊆ V" by (rule greedy_subset_V)
    from fin sub show ?thesis by (rule card_mono)
  qed
  show ?thesis
  proof (cases "i ≤ card V")
    case True
    then show ?thesis using A1 by (simp add: min_def)
  next
    case False
    then show ?thesis using A2 by (simp add: min_def)
  qed
qed
```

Length and endpoints of the intermediate sequence.

```
lemma greedy_sequence_length [simp]:
```

```
  "length (greedy_sequence n) = Suc n"
  by (simp add: greedy_sequence_def)

lemma greedy_sequence_0 [simp]:
  "greedy_sequence n ! 0 = {}"
  using greedy_sequence_nth[of 0 n] by simp

lemma greedy_sequence_last [simp]:
  "greedy_sequence n ! n = greedy_set n"
  using greedy_sequence_nth by simp
```

At a non-empty remainder, the chosen element is new and lies in `V - S`.

```
lemma chosen_in_remainder_nonempty:
  assumes rem_ne: "V - greedy_set i ≠ {}"
  defines x_def:  "x ≡ argmax_gain (greedy_set i) (V - greedy_set i)"
  shows   "x ∈ V - greedy_set i" "x ∉ greedy_set i"
proof -
  have finA: "finite (V - greedy_set i)"
    using finite_V by simp
  have xinA:
    "argmax_gain (greedy_set i) (V - greedy_set i) ∈ V - greedy_set i"
    using argmax_gain_mem[OF finA rem_ne] .
  show "x ∈ V - greedy_set i"
    unfolding x_def by (rule xinA)
  then show "x ∉ greedy_set i" by simp
qed
```

At a non-empty step, `greedy_set (Suc i)` is obtained by inserting the arg-max element into `greedy_set i`. This is often useful in counting arguments.

```
lemma greedy_increment_nonempty[simp]:
  assumes rem_ne: "V - greedy_set i ≠ {}"
  shows   "greedy_set (Suc i) =
           insert (argmax_gain (greedy_set i) (V - greedy_set i)) (greedy_set
i)"
proof -
  have not_subset: "¬ V ⊆ greedy_set i"
    using rem_ne by (simp add: Diff_eq_empty_iff)

  have def:
    "greedy_set (Suc i) =
       (let S = greedy_set i in
        if V ⊆ S then S else insert (argmax_gain S (V - S)) S)"
    by simp

  show ?thesis
    unfolding def
    using not_subset
    by (simp add: Let_def)
```

qed

One-step update of the objective value along the greedy sequence.

```
lemma greedy_step_f_eq:
  assumes "V - greedy_set i ≠ {}"
  shows
    "f (greedy_set (Suc i)) =
       f (greedy_set i) +
       gain (greedy_set i)
         (argmax_gain (greedy_set i) (V - greedy_set i))"
proof -
  let ?S = "greedy_set i"
  let ?x = "argmax_gain ?S (V - ?S)"

  have gs_Suc:
    "greedy_set (Suc i) =
       insert (argmax_gain (greedy_set i) (V - greedy_set i)) (greedy_set
i)"
    using assms by (rule greedy_increment_nonempty)

  hence "greedy_set (Suc i) = ?S ∪ {?x}"
    by simp

  hence "f (greedy_set (Suc i)) = f (?S ∪ {?x})"
    by simp
  also have "... = f ?S + gain ?S ?x"
    by (simp add: gain_def)
  finally show ?thesis
    by simp
qed

end

context Cardinality_Constraint
begin

interpretation Greedy_Concrete: Greedy_Setup V f k argmax_gain_some
proof
  fix S :: "'a set" and A :: "'a set"
  assume fin: "finite A" and ne: "A ≠ {}"
  show "argmax_gain_some S A ∈ A"
    using argmax_gain_some_mem[OF fin ne] .
next
  fix S :: "'a set" and A :: "'a set"
  assume fin: "finite A" and ne: "A ≠ {}"
  show "∀y∈A. gain S y ≤ gain S (argmax_gain_some S A)"
  proof
    fix y
    assume yA: "y ∈ A"
```

```
    show "gain S y ≤ gain S (argmax_gain_some S A)"
      using argmax_gain_some_max[OF fin ne yA] .
  qed
qed

end

end
theory Lazy_Greedy_Oracle
  imports "Greedy_Submodular_Construct"
begin
```

This theory provides auxiliary lazy-selection primitives based on cached upper bounds for marginal gains. It is used by the stateful lazy greedy construction below, while the final approximation theorem is stated in the proof layer.

2 Lazy selection via cached upper bounds

```
context Submodular_Func
begin

definition ub_valid :: "'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ bool" where
  "ub_valid S A ub ⟷ (∀x∈A. gain S x ≤ ub x)"

definition untight :: "'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ 'a set" where
  "untight S A ub = {x∈A. ub x > gain S x}"

definition tighten :: "'a set ⇒ ('a ⇒ real) ⇒ 'a ⇒ ('a ⇒ real)" where
  "tighten S ub x = ub(x := gain S x)"

definition pick_ub_some :: "'a set ⇒ ('a ⇒ real) ⇒ 'a" where
  "pick_ub_some A ub = (SOME x. x ∈ A ∧ is_arg_max ub (λy. y ∈ A) x)"

lemma pick_ub_some_mem:
  assumes finA: "finite A" and neA: "A ≠ {}"
  shows "pick_ub_some A ub ∈ A"
proof -
  have exB: "∃x∈A. is_arg_max ub (λy. y ∈ A) x"
    using finite_is_arg_max_in[OF finA neA] by blast
  have ex: "∃x. x ∈ A ∧ is_arg_max ub (λy. y ∈ A) x"
    using exB by auto
  show ?thesis
    unfolding pick_ub_some_def
    using someI_ex[OF ex] by blast
qed

lemma pick_ub_some_max:
```

```
  assumes finA: "finite A" and neA: "A ≠ {}" and yA: "y ∈ A"
  shows "ub y ≤ ub (pick_ub_some A ub)"
proof -
  have exB: "∃x∈A. is_arg_max ub (λz. z ∈ A) x"
    using finite_is_arg_max_in[OF finA neA] by blast
  have ex: "∃x. x ∈ A ∧ is_arg_max ub (λz. z ∈ A) x"
    using exB by auto
  have chosen: "is_arg_max ub (λz. z ∈ A) (pick_ub_some A ub)"
    unfolding pick_ub_some_def
    using someI_ex[OF ex] by blast
  show ?thesis
    using is_arg_maxD_le[OF chosen yA] .
qed

fun lazy_argmax_gain_fuel ::
  "nat ⇒ 'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ 'a"
where
  "lazy_argmax_gain_fuel 0 S A ub = pick_ub_some A ub"
| "lazy_argmax_gain_fuel (Suc n) S A ub =
    (let x = pick_ub_some A ub in
     if ub x = gain S x then x
     else lazy_argmax_gain_fuel n S A (tighten S ub x))"

definition lazy_argmax_gain :: "'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ 'a"
where
  "lazy_argmax_gain S A ub = lazy_argmax_gain_fuel (card A) S A ub"

lemma ub_valid_tighten:
  assumes ubv: "ub_valid S A ub"
  shows "ub_valid S A (tighten S ub x)"
  using ubv unfolding ub_valid_def tighten_def by auto

lemma untight_tighten:
  assumes xA: "x ∈ A" and gt: "ub x > gain S x"
  shows "untight S A (tighten S ub x) = untight S A ub - {x}"
  using xA gt
  unfolding untight_def tighten_def
  by auto


lemma finite_untight:
  assumes finA: "finite A"
  shows "finite (untight S A ub)"
  using finA unfolding untight_def by simp

lemma lazy_argmax_gain_fuel_max:
  assumes finA: "finite A" and neA: "A ≠ {}"
  shows "ub_valid S A ub ⟹ card (untight S A ub) ≤ n ⟹
         ∀y∈A. gain S y ≤ gain S (lazy_argmax_gain_fuel n S A ub)"
```

```
proof (induction n arbitrary: ub)
  case 0
  from 0 have ubv: "ub_valid S A ub" by simp
  from 0 have bound: "card (untight S A ub) ≤ 0" by simp

  have finU: "finite (untight S A ub)" using finite_untight[OF finA] .
  have U0: "untight S A ub = {}"
    using bound finU by auto

  have ub_le_gain: "∀y∈A. ub y ≤ gain S y"
  proof (intro ballI)
    fix y assume yA: "y ∈ A"
    have "¬ (ub y > gain S y)"
      using U0 yA unfolding untight_def by auto
    thus "ub y ≤ gain S y" by simp
  qed

  have ub_eq_gain: "∀y∈A. ub y = gain S y"
  proof (intro ballI)
    fix y assume yA: "y ∈ A"
    have "gain S y ≤ ub y"
      using ubv yA unfolding ub_valid_def by auto
    moreover have "ub y ≤ gain S y"
      using ub_le_gain yA by auto
    ultimately show "ub y = gain S y" by simp
  qed

  let ?x = "pick_ub_some A ub"
  have xA: "?x ∈ A" using pick_ub_some_mem[OF finA neA] .
  have ubx: "ub ?x = gain S ?x"
    using ub_eq_gain xA by auto

  show ?case
  proof (intro ballI)
    fix y assume yA: "y ∈ A"
    have "gain S y = ub y" using ub_eq_gain yA by auto
    also have "... ≤ ub ?x"
      using pick_ub_some_max[OF finA neA yA] .
    also have "... = gain S ?x"
      using ubx by simp
    finally show "gain S y ≤ gain S (lazy_argmax_gain_fuel 0 S A ub)"
      by (simp add: Let_def)
  qed
next
  case (Suc n ub)
  from Suc.prems have ubv: "ub_valid S A ub" by simp
  from Suc.prems have bound: "card (untight S A ub) ≤ Suc n" by simp

  let ?x = "pick_ub_some A ub"
```

```
  have xA: "?x ∈ A" using pick_ub_some_mem[OF finA neA] .

  show ?case
  proof (cases "ub ?x = gain S ?x")
    case True
    show ?thesis
    proof (intro ballI)
      fix y assume yA: "y ∈ A"
      have "gain S y ≤ ub y"
        using ubv yA unfolding ub_valid_def by auto
      also have "... ≤ ub ?x"
        using pick_ub_some_max[OF finA neA yA] .
      finally show "gain S y ≤ gain S (lazy_argmax_gain_fuel (Suc n) S
A ub)"
        using True by (simp add: Let_def)
    qed
  next
    case False
    have le_gx: "gain S ?x ≤ ub ?x"
      using ubv xA unfolding ub_valid_def by auto
    have gt: "ub ?x > gain S ?x"
      using le_gx False by (meson eq_iff not_le order_le_less)

    have Ueq: "untight S A (tighten S ub ?x) = untight S A ub - {?x}"
      using xA gt by (simp add: untight_tighten)

    have xU: "?x ∈ untight S A ub"
      using xA gt unfolding untight_def by auto

    have finU: "finite (untight S A ub)"
      using finite_untight[OF finA] .

    have bound': "card (untight S A (tighten S ub ?x)) ≤ n"
    proof -
      have "card (untight S A (tighten S ub ?x)) = card (untight S A ub
- {?x})"
        by (simp add: Ueq)
      also have "... = card (untight S A ub) - 1"
        using finU xU by (simp add: card_Diff_singleton)
      finally show ?thesis
        using bound by arith
    qed

    have IH: "∀y∈A. gain S y ≤ gain S (lazy_argmax_gain_fuel n S A (tighten
S ub ?x))"
      using Suc.IH[OF ub_valid_tighten[OF ubv] bound'] .

    show ?thesis
      using False IH by (simp add: Let_def)
```

```
  qed
qed

lemma lazy_argmax_gain_fuel_mem:
  assumes finA: "finite A" and neA: "A ≠ {}"
  shows "lazy_argmax_gain_fuel n S A ub ∈ A"
proof (induction n arbitrary: ub)
  case 0
  show ?case
    using pick_ub_some_mem[OF finA neA] by simp
next
  case (Suc n)
  let ?x = "pick_ub_some A ub"
  have xA: "?x ∈ A"
    using pick_ub_some_mem[OF finA neA] .

  show ?case
  proof (cases "ub ?x = gain S ?x")
    case True
    then show ?thesis
      using xA by (simp add: Let_def)
  next
    case False
    then show ?thesis
      using Suc.IH[of "tighten S ub ?x"] finA neA
      by (simp add: Let_def)
  qed
qed

lemma lazy_argmax_gain_mem:
  assumes finA: "finite A" and neA: "A ≠ {}"
  shows "lazy_argmax_gain S A ub ∈ A"
  unfolding lazy_argmax_gain_def
  by (rule lazy_argmax_gain_fuel_mem[OF finA neA])

lemma lazy_argmax_gain_max:
  assumes finA: "finite A" and neA: "A ≠ {}"
      and ubv: "ub_valid S A ub"
  shows "∀y∈A. gain S y ≤ gain S (lazy_argmax_gain S A ub)"
proof -
  have finU: "finite (untight S A ub)"
    using finite_untight[OF finA] .
  have "card (untight S A ub) ≤ card A"
    using card_mono[OF finA] unfolding untight_def by auto
  hence "∀y∈A. gain S y ≤ gain S (lazy_argmax_gain_fuel (card A) S A
ub)"
    using lazy_argmax_gain_fuel_max[OF finA neA ubv] by blast
  thus ?thesis
    unfolding lazy_argmax_gain_def .
```

```
qed

end

end
theory Lazy_Greedy_Stateful
  imports Lazy_Greedy_Oracle
begin
```

3 Stateful lazy greedy with cached upper bounds

```
record 'a lg_state =
  Sg  :: "'a set"
  ubg :: "'a ⇒ real"

context Cardinality_Constraint
begin
```

3.1 State: selected set and cached upper bounds

```
definition init_ub :: "'a ⇒ real" where
  "init_ub x = gain {} x"

definition init_state :: "'a lg_state" where
  "init_state = ⦇ Sg = {}, ubg = init_ub ⦈"

definition remaining :: "'a lg_state ⇒ 'a set" where
  "remaining st = V - Sg st"
```

3.2 Inner lazy selection returning updated upper bounds

```
fun lazy_argmax_gain_fuel_state ::
  "nat ⇒ 'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ ('a × ('a ⇒ real))"
where
  "lazy_argmax_gain_fuel_state 0 S A ub =
     (let x = pick_ub_some A ub in (x, ub))"
| "lazy_argmax_gain_fuel_state (Suc n) S A ub =
     (let x = pick_ub_some A ub in
      if ub x = gain S x then (x, ub)
      else lazy_argmax_gain_fuel_state n S A (tighten S ub x))"

definition lazy_select :: "'a set ⇒ 'a set ⇒ ('a ⇒ real) ⇒ ('a × ('a
⇒ real))" where
  "lazy_select S A ub = lazy_argmax_gain_fuel_state (card A) S A ub"

lemma lazy_argmax_gain_fuel_state_fst:
  "fst (lazy_argmax_gain_fuel_state n S A ub) = lazy_argmax_gain_fuel
n S A ub"
proof (induction n arbitrary: ub)
```

```
  case 0
  then show ?case by (simp add: Let_def)
next
  case (Suc n)
  then show ?case
    by (simp add: Let_def)
qed

lemma lazy_select_fst:
  "fst (lazy_select S A ub) = lazy_argmax_gain S A ub"
  unfolding lazy_select_def lazy_argmax_gain_def
  using lazy_argmax_gain_fuel_state_fst[of "card A" S A ub]
  by simp

lemma lazy_argmax_gain_fuel_state_ub_valid:
  assumes ubv: "ub_valid S A ub"
  shows "ub_valid S A (snd (lazy_argmax_gain_fuel_state n S A ub))"
  using ubv
proof (induction n arbitrary: ub)
  case 0
  show ?case
    using 0 by (simp add: Let_def)
next
  case (Suc n ub)
  from Suc.prems have ubv_current: "ub_valid S A ub" by simp

  let ?x = "pick_ub_some A ub"

  show ?case
  proof (cases "ub ?x = gain S ?x")
    case True
    then show ?thesis
      using ubv_current by (simp add: Let_def)
  next
    case False
    have ubv_tight: "ub_valid S A (tighten S ub ?x)"
      using ub_valid_tighten[OF ubv_current] .

    have IH_result: "ub_valid S A (snd (lazy_argmax_gain_fuel_state n
S A (tighten S ub ?x)))"
      using Suc.IH[OF ubv_tight] .

    show ?thesis
      using False IH_result by (simp add: Let_def)
  qed
qed

lemma lazy_select_ub_valid:
  assumes ubv: "ub_valid S A ub"
```

```
  shows "ub_valid S A (snd (lazy_select S A ub))"
  unfolding lazy_select_def
  using lazy_argmax_gain_fuel_state_ub_valid[OF ubv] .
```

3.3 Preservation of upper-bound validity across outer iterations

```
lemma ub_valid_init:
  "ub_valid {} V init_ub"
  unfolding ub_valid_def init_ub_def
  by simp

lemma ub_valid_after_insert:
  assumes ubv: "ub_valid S (V - S) ub"
      and Ssub: "S ⊆ V"
      and x_in: "x ∈ V - S"
  shows "ub_valid (insert x S) (V - insert x S) ub"
proof (unfold ub_valid_def, intro ballI)
  fix y assume yR: "y ∈ V - insert x S"
  have yV: "y ∈ V" and y_notT: "y ∉ insert x S"
    using yR by auto

  have y_in_old: "y ∈ V - S"
    using yR by auto

  have TsubV: "insert x S ⊆ V"
    using Ssub x_in by auto

  have dec_ge: "gain S y ≥ gain (insert x S) y"
    using gain_decreasing[OF _ TsubV yV y_notT] Ssub
    by auto

  have dec: "gain (insert x S) y ≤ gain S y"
    using dec_ge by linarith

  have up: "gain S y ≤ ub y"
    using ubv y_in_old unfolding ub_valid_def by auto

  show "gain (insert x S) y ≤ ub y"
    using dec up by linarith
qed
```

3.4 One outer step and the full stateful algorithm

```
definition lazy_step :: "'a lg_state ⇒ 'a lg_state" where
  "lazy_step st =
     (let S = Sg st;
          A = remaining st;
          ub = ubg st
```

```
      in if A = {} then st
         else
           (let p = lazy_select S A ub;
                x = fst p;
                ub' = snd p
            in (| Sg = insert x S, ubg = ub' |)))"

lemma lazy_step_nonempty_Sg:
  assumes rem_ne: "remaining st ≠ {}"
  shows
    "Sg (lazy_step st) =
      insert (fst (lazy_select (Sg st) (remaining st) (ubg st))) (Sg st)"
  using rem_ne
  unfolding lazy_step_def
  by (simp add: Let_def)

lemma lazy_step_nonempty_ubg:
  assumes rem_ne: "remaining st ≠ {}"
  defines "p ≡ lazy_select (Sg st) (remaining st) (ubg st)"
  shows "ubg (lazy_step st) = snd p"
  using rem_ne
  unfolding lazy_step_def p_def
  by (simp add: Let_def)

fun lazy_state :: "nat ⇒ 'a lg_state" where
  "lazy_state 0 = init_state"
| "lazy_state (Suc i) = lazy_step (lazy_state i)"

definition lazy_set :: "nat ⇒ 'a set" where
  "lazy_set i = Sg (lazy_state i)"
```

3.5 Main invariants: subset property and validity on the remaining set

```
lemma lazy_step_idle:
  assumes "remaining st = {}"
  shows "lazy_step st = st"
  using assms
  unfolding lazy_step_def remaining_def
  by (simp add: Let_def)


lemma lazy_state_subset_V:
  "Sg (lazy_state i) ⊆ V"
proof (induction i)
  case 0
  then show ?case
    by (simp add: init_state_def)
next
```

```
case (Suc i)
let ?st = "lazy_state i"
have IH: "Sg ?st ⊆ V" using Suc.IH .

show ?case
proof (cases "remaining ?st = {}")
  case True
  have step_idle: "lazy_step ?st = ?st"
    using lazy_step_idle[OF True] .
  show ?thesis
    using IH by (simp add: step_idle)
next
  case False
  let ?S  = "Sg ?st"
  let ?A  = "remaining ?st"
  let ?ub = "ubg ?st"
  let ?p  = "lazy_select ?S ?A ?ub"
  let ?x  = "fst ?p"

  have A_subV: "?A ⊆ V"
    unfolding remaining_def using IH by auto

  have finA: "finite ?A"
    by (rule finite_subset[OF A_subV finite_V])

  have neA: "?A ≠ {}"
    using False by simp

  have x_inA: "?x ∈ ?A"
  proof -
    have x_eq: "?x = lazy_argmax_gain ?S ?A ?ub"
      using lazy_select_fst by simp
    show ?thesis
      unfolding x_eq
      using lazy_argmax_gain_mem[OF finA neA] .
  qed

  have xV: "?x ∈ V"
    using A_subV x_inA by auto

  have Sg_step:
    "Sg (lazy_step ?st) = insert ?x ?S"
    using lazy_step_nonempty_Sg[OF False]
    by simp

  show ?thesis
    using IH xV
    by (auto simp: Sg_step)
qed
```

```
qed

lemma lazy_state_ub_valid:
  "ub_valid (Sg (lazy_state i)) (remaining (lazy_state i)) (ubg (lazy_state
i))"
proof (induction i)
  case 0
  show ?case
    unfolding lazy_state.simps init_state_def remaining_def
    using ub_valid_init by simp
next
  case (Suc i)
  let ?st = "lazy_state i"
  have IH: "ub_valid (Sg ?st) (remaining ?st) (ubg ?st)"
    by (rule Suc.IH)

  show ?case
  proof (cases "remaining ?st = {}")
    case True
    have "lazy_step ?st = ?st"
      using lazy_step_idle[OF True] .
    then show ?thesis
      using IH by simp
  next
    case False
    let ?S = "Sg ?st"
    let ?A = "remaining ?st"
    let ?ub = "ubg ?st"
    let ?p = "lazy_select ?S ?A ?ub"
    let ?x = "fst ?p"
    let ?ub' = "snd ?p"

    have ubvA: "ub_valid ?S ?A ?ub"
      using IH .
    have ubvA': "ub_valid ?S ?A ?ub'"
      using lazy_select_ub_valid[OF ubvA] by simp

    have SsubV: "?S ⊆ V"
      using lazy_state_subset_V[of i] by simp

    have A_def: "?A = V - ?S"
      unfolding remaining_def by simp

    have finA: "finite ?A"
      unfolding remaining_def
      using finite_V
      by simp

    have neA: "?A ≠ {}"
```

```
    using False by simp

  have x_inA: "?x ∈ ?A"
    using lazy_argmax_gain_mem[OF finA neA]
    by (simp add: lazy_select_fst)

  have ubvVS: "ub_valid ?S (V - ?S) ?ub'"
    using ubvA' by (simp add: A_def)

  have x_in_old: "?x ∈ V - ?S"
    using x_inA by (simp add: A_def)

  have ubv_next: "ub_valid (insert ?x ?S) (V - insert ?x ?S) ?ub'"
    using ub_valid_after_insert[OF ubvVS SsubV x_in_old] .

  have Sg_next: "Sg (lazy_step ?st) = insert ?x ?S"
    using lazy_step_nonempty_Sg[OF False] by simp

  have ubg_next: "ubg (lazy_step ?st) = ?ub'"
    using lazy_step_nonempty_ubg[OF False] by (simp add: Let_def)

  have rem_next: "remaining (lazy_step ?st) = V - insert ?x ?S"
    unfolding remaining_def Sg_next by simp

  show ?thesis
    unfolding lazy_state.simps Sg_next ubg_next rem_next
    using ubv_next by simp
 qed
qed
```

3.6 Correctness of the lazy greedy step

```
lemma lazy_step_is_argmax:
  assumes rem_ne: "remaining st ≠ {}"
      and ubv: "ub_valid (Sg st) (remaining st) (ubg st)"
      and finA: "finite (remaining st)"
  defines "p ≡ lazy_select (Sg st) (remaining st) (ubg st)"
  defines "x ≡ fst p"
  shows "∀y∈remaining st. gain (Sg st) y ≤ gain (Sg st) x"
proof -
  have x_eq: "x = lazy_argmax_gain (Sg st) (remaining st) (ubg st)"
    unfolding x_def p_def using lazy_select_fst by simp
  show ?thesis
    unfolding x_eq
    using lazy_argmax_gain_max[OF finA rem_ne ubv] .
qed

end
```

```
end
theory Greedy_Step_Spec
  imports
    "../Algorithms/Greedy_Submodular_Construct"
begin
```

Step-specification interface for greedy-style algorithms.

The main construction locale is *Greedy_Setup*. The following locale is an intentionally thin named view of this setup, using the name *select* for an oracle that chooses a maximum-marginal-gain element from every finite non-empty candidate set. This keeps later corollaries independent of the concrete choice-based oracle used for the basic greedy construction.

```
locale Greedy_Step_Oracle =
  Greedy_Setup V f k select
  for V :: "'a set"
    and f :: "'a set ⇒ real"
    and k :: nat
    and select :: "'a set ⇒ 'a set ⇒ 'a"
begin

end

end
theory Greedy_Submodular_Approx
  imports
    Greedy_Step_Spec
begin
```

We first derive analytic bounds for the coefficient *1 - (1 - 1/k)^k* appearing in the Nemhauser–Wolsey approximation ratio. These bounds are later combined with the greedy gap recurrence to obtain the standard *1 - 1/exp 1* guarantee.

First, we relate the finite quantity *(1 - 1/k)^k* to the exponential function via a standard exponential inequality.

```
lemma pow_one_minus_inv_le_exp_neg1:
  fixes k :: nat
  assumes "k ≥ 1"
  shows "(1 - 1 / real k) ^ k ≤ exp (-1 :: real)"
proof -
  have f1: "0 < k"
    using assms by auto
  have "1 ≤ real k"
    using assms one_of_nat_le_iff by blast
  then show ?thesis
    using f1 exp_ge_one_minus_x_over_n_power_n by presburger
qed
```

As a corollary, we obtain a uniform lower bound *1 - (1 - 1/k)^k ≥ 1*

- *1/exp 1* for all *k ≥ 1*.

lemma *coeff_ge_1_minus_inv_exp:*
  fixes *k :: nat*
  assumes *"k ≥ 1"*
  shows *"1 - (1 - 1 / real k) ^ k ≥ 1 - 1 / exp 1"*
proof -
  from *pow_one_minus_inv_le_exp_neg1[OF assms]*
  have *"(1 - 1 / real k) ^ k ≤ exp (-1 :: real)"* .
  then have *"1 - (1 - 1 / real k) ^ k ≥ 1 - exp (-1 :: real)"*
    by *simp*
  also have *"exp (-1 :: real) = 1 / exp 1"*
    by *(simp add: exp_minus field_simps)*
  finally show *?thesis* .
qed

context *Greedy_Setup*
begin

4 Greedy approximation for monotone submodular maximization

In this section we formalize the classical Nemhauser–Wolsey guarantee: for a non-negative, monotone, submodular function on a finite ground set, the greedy algorithm that selects *k* elements achieves at least *1 - (1 - 1/k)^k* times the optimal value. Combining this with the analytic bound above yields the familiar *1 - 1/e* approximation factor.

Elementary algebraic identity used in the gap recurrence.

lemma *one_minus_inv_times:*
  fixes *x :: real*
  shows *"(1 - 1 / real k) * x = x - x / real k"*
  by *(simp add: left_diff_distrib)*

4.1 Greedy gap analysis

We use the problem-level optimal value and the reusable marginal-gain averaging lemma from the base cardinality context to derive the greedy gap recurrence.

4.1.1 Gap sequence

We introduce the gap sequence *gap i = OPT_k - f (greedy_set i)* and show that it satisfies a simple linear recurrence under the greedy update.

definition *gap :: "nat ⇒ real"* **where**
  "gap i = OPT_k - f (greedy_set i)"

One-step inequality: the marginal gain of the greedy choice lower-bounds the average improvement towards *OPT_k*.

```
lemma greedy_step_ineq:
  assumes "i < k"
    and S_sub: "greedy_set i ⊆ V"
    and R_nonempty: "V - greedy_set i ≠ {}"
  shows "gain (greedy_set i)
           (argmax_gain (greedy_set i) (V - greedy_set i))
         ≥ (OPT_k - f (greedy_set i)) / real k"
proof -
  let ?S = "greedy_set i"
  let ?R = "V - ?S"

  obtain X where X_feas: "feasible X" and X_opt: "f X = OPT_k"
    using exists_opt_set by blast
  from X_feas have X_sub: "X ⊆ V" and cardX_le_k: "card X ≤ k"
    unfolding feasible_def by auto

  have S_sub': "?S ⊆ V"
    using S_sub .

  have cardS_lt_k: "card ?S < k"
  proof -
    have "card ?S ≤ i"
      by (rule card_greedy_le_i)
    with assms(1) show ?thesis
      by (meson le_less_trans)
  qed

  from marginal_gain_lower_bound[
       OF S_sub' X_sub cardS_lt_k cardX_le_k]
  obtain e where e_inR: "e ∈ V - ?S"
    and e_lb: "gain ?S e ≥ (f X - f ?S) / real k"
    by blast

  have finV: "finite V"
    by (rule finite_V)

  have finR: "finite ?R"
    using finV by auto
  have R_nonempty': "?R ≠ {}"
    using R_nonempty by simp

  have argmax_max:
    "∀y∈?R. gain ?S y ≤ gain ?S (argmax_gain ?S ?R)"
    using argmax_gain_max[OF finR R_nonempty'] .

  from argmax_max e_inR
  have e_le_argmax:
```

```
    "gain ?S e ≤ gain ?S (argmax_gain ?S ?R)"
    by auto

  have argmax_lb:
    "gain ?S (argmax_gain ?S ?R)
       ≥ (f X - f ?S) / real k"
    using e_lb e_le_argmax by linarith

  have "(f X - f ?S) / real k = (OPT_k - f ?S) / real k"
    using X_opt by simp

  with argmax_lb
  have "gain ?S (argmax_gain ?S ?R)
        ≥ (OPT_k - f ?S) / real k"
    by simp

  thus ?thesis
    by simp
qed
```

Greedy sets are feasible whenever their size is at most k.

```
lemma greedy_set_feasible:
  assumes S_sub: "greedy_set i ⊆ V"
      and card_le_i: "card (greedy_set i) ≤ i"
      and i_le_k: "i ≤ k"
  shows "feasible (greedy_set i)"
proof -
  have "card (greedy_set i) ≤ k"
    using card_le_i i_le_k by (meson order_trans)
  with S_sub show ?thesis
    unfolding feasible_def by auto
qed

corollary greedy_feasible:
  assumes "i ≤ k"
  shows "feasible (greedy_set i)"
  using greedy_set_feasible[OF greedy_subset_V card_greedy_le_i assms]
.
```

The gap is non-negative along the greedy sequence.

```
lemma gap_nonneg:
  assumes S_sub: "greedy_set i ⊆ V"
      and card_le_i: "card (greedy_set i) ≤ i"
      and i_le_k: "i ≤ k"
  shows "0 ≤ gap i"
proof -
  have S_feas: "feasible (greedy_set i)"
    using greedy_set_feasible[OF S_sub card_le_i i_le_k] .
  have "f (greedy_set i) ≤ OPT_k"
```

```
    using OPT_k_upper_bound[OF S_feas] .
  then have "0 ≤ OPT_k - f (greedy_set i)"
    by simp
  thus ?thesis
    unfolding gap_def by simp
qed

corollary greedy_gap_nonneg:
  assumes "i ≤ k"
  shows "0 ≤ gap i"
  using gap_nonneg[OF greedy_subset_V card_greedy_le_i assms] .

corollary greedy_remainder_nonempty:
  assumes "i < k"
  shows "V - greedy_set i ≠ {}"
proof -
  have "card (greedy_set i) ≤ i"
    by (rule card_greedy_le_i)
  also have "... < k"
    using assms by simp
  also have "... ≤ card V"
    using k_le_cardV by simp
  finally have "card (greedy_set i) < card V" .
  thus ?thesis
    by (rule remainder_nonempty_if_card_ltV)
qed
```

Gap recurrence: each step reduces the gap by at least a `1/k` fraction.

```
lemma gap_step_diff:
  assumes "i < k"
    and S_sub: "greedy_set i ⊆ V"
    and R_nonempty: "V - greedy_set i ≠ {}"
  shows "gap (Suc i) ≤ gap i - gap i / real k"
proof -
  let ?S = "greedy_set i"

  have base:
    "gain ?S (argmax_gain ?S (V - ?S))
       ≥ (OPT_k - f ?S) / real k"
    using greedy_step_ineq[OF assms] by simp

  have gap_Suc_eq:
    "gap (Suc i)
       = OPT_k - f ?S
         - gain ?S (argmax_gain ?S (V - ?S))"
  proof -
    have "gap (Suc i) = OPT_k - f (greedy_set (Suc i))"
      by (simp add: gap_def)
    also have "... = OPT_k
```

```
                - (f ?S
                   + gain ?S (argmax_gain ?S (V - ?S)))"
      using greedy_step_f_eq[OF R_nonempty] by simp
    also have "... = OPT_k - f ?S
                   - gain ?S (argmax_gain ?S (V - ?S))"
      by simp
    finally show ?thesis .
  qed

  have "OPT_k - f ?S
        - gain ?S (argmax_gain ?S (V - ?S))
      ≤ OPT_k - f ?S
        - (OPT_k - f ?S) / real k"
    using base by linarith
  hence "gap (Suc i)
         ≤ OPT_k - f ?S - (OPT_k - f ?S) / real k"
    using gap_Suc_eq by simp

  also have "OPT_k - f ?S - (OPT_k - f ?S) / real k
             = gap i - gap i / real k"
    by (simp add: gap_def)

  finally show ?thesis .
qed
```

In multiplicative form, the gap shrinks by a factor of at most `1 - 1/real k` per step.

```
lemma gap_step:
  assumes "i < k"
    and "greedy_set i ⊆ V"
    and "V - greedy_set i ≠ {}"
  shows "gap (Suc i) ≤ (1 - 1 / real k) * gap i"
proof -
  have "gap (Suc i) ≤ gap i - gap i / real k"
    using gap_step_diff[OF assms] .
  also have "gap i - gap i / real k
             = (1 - 1 / real k) * gap i"
    using one_minus_inv_times[of "gap i"] by simp
  finally show ?thesis .
qed

lemma gap_0[simp]: "gap 0 = OPT_k"
proof -
  have "gap 0 = OPT_k - f (greedy_set 0)"
    by (simp add: gap_def)
  also have "greedy_set 0 = {}" by simp
  also have "f {} = 0" by (rule f_empty)
  finally show ?thesis by simp
qed
```

Geometric decay of the gap: after `i` greedy steps the remaining gap is bounded by `(1 - 1/real k)^i * OPT_k`.

```
lemma gap_geometric:
  assumes k_pos: "k > 0"
    and i_le_k: "i ≤ k"
  shows "gap i ≤ (1 - 1 / real k) ^ i * OPT_k"
using i_le_k
proof (induction i)
  case 0
  have "gap 0 = OPT_k" by simp
  thus ?case by simp
next
  case (Suc i)
  have i_le_k: "i ≤ k" using Suc.prems by simp
  have i_lt_k: "i < k" using Suc.prems by simp

  have S_sub: "greedy_set i ⊆ V"
    by (rule greedy_subset_V)

  have cardSi_lt_V: "card (greedy_set i) < card V"
  proof -
    have "card (greedy_set i) ≤ i"
      by (rule card_greedy_le_i)
    also have "... < k" using i_lt_k by simp
    also have "... ≤ card V" using k_le_cardV by simp
    finally show ?thesis .
  qed

  have R_nonempty: "V - greedy_set i ≠ {}"
    using remainder_nonempty_if_card_ltV[OF cardSi_lt_V] .

  have step:
    "gap (Suc i) ≤ (1 - 1 / real k) * gap i"
    using gap_step[OF i_lt_k S_sub R_nonempty] .

  have coef_nonneg: "0 ≤ (1 - 1 / real k)"
  proof -
    have "1 ≤ real k" using k_pos by simp
    then have "1 / real k ≤ 1"
      by (simp add: field_simps)
    then show ?thesis
      by simp
  qed

  have mult_mono:
    "(1 - 1 / real k) * gap i
      ≤ (1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)"
  proof -
    have IH: "gap i ≤ (1 - 1 / real k) ^ i * OPT_k"
```

```
    using Suc.IH i_le_k by simp
  from IH coef_nonneg show ?thesis
    by (rule mult_left_mono)
qed

have pow_Suc:
 "(1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)
   = (1 - 1 / real k) ^ Suc i * OPT_k"
 by (simp add: mult_ac)

from step mult_mono have
  "gap (Suc i)
     ≤ (1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)"
  by (rule order_trans)
hence "gap (Suc i)
       ≤ (1 - 1 / real k) ^ Suc i * OPT_k"
  by (simp add: pow_Suc)
thus ?case
  by simp
qed
```

As a consequence, the value of the greedy solution after `k` steps is at least `(1 - (1 - 1/real k)^k) * OPT_k`.

```
lemma greedy_sequence_bound:
  assumes k_pos: "k > 0"
  shows "f (greedy_set k) ≥ (1 - (1 - 1 / real k) ^ k) * OPT_k"
proof -
  have gap_bound:
    "gap k ≤ (1 - 1 / real k) ^ k * OPT_k"
    using gap_geometric[OF k_pos le_refl] .

  have f_eq:
    "f (greedy_set k) = OPT_k - gap k"
    by (simp add: gap_def)

  have lower_bound:
    "OPT_k - gap k ≥ OPT_k - (1 - 1 / real k) ^ k * OPT_k"
  proof -
    have "- gap k ≥ - ((1 - 1 / real k) ^ k * OPT_k)"
      using gap_bound by simp
    thus ?thesis
      by simp
  qed

  have "f (greedy_set k) ≥ OPT_k - (1 - 1 / real k) ^ k * OPT_k"
    using f_eq lower_bound by simp

  also have "OPT_k - (1 - 1 / real k) ^ k * OPT_k
             = (1 - (1 - 1 / real k) ^ k) * OPT_k"
```

```
  proof -
    have "OPT_k - (1 - 1 / real k) ^ k * OPT_k
          = 1 * OPT_k - (1 - 1 / real k) ^ k * OPT_k"
      by simp
    also have "... = (1 - (1 - 1 / real k) ^ k) * OPT_k"
      by (simp add: left_diff_distrib)
    finally show ?thesis .
  qed

  finally show ?thesis .
qed
```

4.2 Non-negativity of OPT and approximation ratio

Combining the discrete bound with the analytic inequality for `1 - (1 - 1/k)^k` yields the standard `1 - 1/e` approximation factor.

```
theorem greedy_approximation:
  assumes k_pos: "k > 0"
  shows "f (greedy_set k) ≥ (1 - 1 / exp 1) * OPT_k"
proof -
  have base_bound:
    "f (greedy_set k) ≥ (1 - (1 - 1 / real k) ^ k) * OPT_k"
    using greedy_sequence_bound[OF k_pos] .

  have k_ge1: "k ≥ 1"
    using k_pos by simp

  have coeff_bound:
    "1 - (1 - 1 / real k) ^ k ≥ 1 - 1 / exp 1"
    using coeff_ge_1_minus_inv_exp[OF k_ge1] .

  have nonneg: "0 ≤ OPT_k"
    by (rule OPT_k_nonneg)

  have coeff_mono:
    "(1 - (1 - 1 / real k) ^ k) * OPT_k
      ≥ (1 - 1 / exp 1) * OPT_k"
    using coeff_bound nonneg
    by (rule mult_right_mono)

  from base_bound coeff_mono
  have "f (greedy_set k) ≥ (1 - 1 / exp 1) * OPT_k"
    by (meson order_trans)
  thus ?thesis .
qed
```

4.3 Corollaries

Define the approximation ratio of the greedy algorithm for a given *k* (with the convention that the ratio is *1* when *OPT_k = 0*), and show that it is always at least *1 - 1/exp 1*.

```
definition greedy_ratio :: real where
  "greedy_ratio = (if OPT_k = 0 then 1 else f (greedy_set k) / OPT_k)"

corollary greedy_ratio_ge_1_minus_inv_exp:
  assumes "k > 0"
  shows "greedy_ratio ≥ 1 - 1 / exp 1"
proof (cases "OPT_k = 0")
  case True
  then show ?thesis
    unfolding greedy_ratio_def
    by simp
next
  case False
  then have OPT_pos: "OPT_k > 0"
    using OPT_k_nonneg by auto
  have main: "f (greedy_set k) ≥ (1 - 1 / exp 1) * OPT_k"
    using greedy_approximation[OF assms] .
  show ?thesis
    unfolding greedy_ratio_def
    using main False OPT_pos
    by (simp add: field_simps)
qed

end
```

5 Step-spec corollary

Since *Greedy_Step_Oracle* is a named instance of *Greedy_Setup*, the Nemhauser–Wolsey approximation guarantee transfers directly to any oracle satisfying the step-specification assumptions.

```
context Greedy_Step_Oracle
begin

theorem greedy_step_oracle_approximation:
  assumes "k > 0"
  shows
    "f (Greedy_Setup.greedy_set V select k)
       ≥ (1 - 1 / exp 1) * OPT_k"
  using greedy_approximation[OF assms] .

end

end
```

theory *Lazy_Greedy_Stateful_StepSpec*
  imports *"../Algorithms/Lazy_Greedy_Stateful"*
begin

Sequence-level lemmas for the verified stateful lazy run.

This theory packages the per-iteration facts needed by the approximation proof, such as the membership and maximal-gain properties of the chosen lazy element at step `i`, together with the update equation for the next lazy set.

It is not formalized by instantiating the stateless greedy step-oracle locale. Instead, it exposes the corresponding properties of the concrete verified run.

context *Cardinality_Constraint*
begin

abbreviation *st_i :: "nat ⇒ 'a lg_state"* **where**
  "st_i i ≡ lazy_state i"

abbreviation *S_i :: "nat ⇒ 'a set"* **where**
  "S_i i ≡ lazy_set i"

abbreviation *A_i :: "nat ⇒ 'a set"* **where**
  "A_i i ≡ remaining (st_i i)"

lemma *A_i_eq: "A_i i = V - S_i i"*
  by *(simp add: remaining_def lazy_set_def)*

definition *lazy_choice :: "nat ⇒ 'a"* **where**
  "lazy_choice i = fst (lazy_select (S_i i) (A_i i) (ubg (st_i i)))"

lemma *A_i_finite: "finite (A_i i)"*
proof -
  have *Ssub: "Sg (st_i i) ⊆ V"*
    using *lazy_state_subset_V[of i]* .
  have *Asub: "A_i i ⊆ V"*
    by *(simp add: A_i_eq lazy_set_def Ssub)*
  show *?thesis*
    using *finite_subset[OF Asub finite_V]* .
qed

lemma *lazy_choice_mem:*
  assumes *ne: "A_i i ≠ {}"*
  shows *"lazy_choice i ∈ A_i i"*
proof -
  have *finA: "finite (A_i i)"* **by** *(rule A_i_finite)*
  have *eq: "lazy_choice i = lazy_argmax_gain (S_i i) (A_i i) (ubg (st_i i))"*
    by *(simp add: lazy_choice_def lazy_select_fst)*
  show *?thesis*

```
    unfolding eq
    using lazy_argmax_gain_mem[OF finA ne] .
qed

lemma lazy_choice_max:
  assumes ne: "A_i i ≠ {}"
  shows "∀y∈A_i i. gain (S_i i) y ≤ gain (S_i i) (lazy_choice i)"
proof -
  have finA: "finite (A_i i)"
    by (rule A_i_finite)

  have ubv: "ub_valid (S_i i) (A_i i) (ubg (st_i i))"
  proof -
    have ubvR: "ub_valid (Sg (st_i i)) (remaining (st_i i)) (ubg (st_i
i))"
      using lazy_state_ub_valid[of i] by simp
    show ?thesis
      using ubvR
      by (simp add: A_i_eq lazy_set_def remaining_def)
  qed

  have max_arg:
    "∀y∈A_i i.
       gain (S_i i) y ≤ gain (S_i i) (lazy_argmax_gain (S_i i) (A_i i)
(ubg (st_i i)))"
    using lazy_argmax_gain_max[OF finA ne ubv] .

  show ?thesis
    using max_arg
    by (simp add: lazy_choice_def lazy_select_fst)
qed

lemma lazy_set_Suc_insert:
  assumes ne: "A_i i ≠ {}"
  shows "S_i (Suc i) = insert (lazy_choice i) (S_i i)"
proof -
  have rem_ne: "remaining (st_i i) ≠ {}" using ne by simp
  have Sg_step:
    "Sg (lazy_step (st_i i)) =
      insert (fst (lazy_select (Sg (st_i i)) (remaining (st_i i)) (ubg
(st_i i))))
             (Sg (st_i i))"
    using lazy_step_nonempty_Sg[OF rem_ne] .
  show ?thesis
    by (simp add: lazy_set_def lazy_choice_def Sg_step)
qed

lemma lazy_choice_in_V_minus_S:
  assumes "V - lazy_set i ≠ {}"
```

```
  shows "lazy_choice i ∈ V - lazy_set i"
  using lazy_choice_mem[of i] assms
  by (simp add: A_i_eq)

lemma lazy_choice_argmax_V_minus_S:
  assumes "V - lazy_set i ≠ {}"
  shows "∀y∈V - lazy_set i. gain (lazy_set i) y ≤ gain (lazy_set i)
(lazy_choice i)"
  using lazy_choice_max[of i] assms
  by (simp add: A_i_eq)

lemma lazy_set_Suc_insert_V_minus_S:
  assumes "V - lazy_set i ≠ {}"
  shows "lazy_set (Suc i) = insert (lazy_choice i) (lazy_set i)"
  using lazy_set_Suc_insert[of i] assms
  by (simp add: A_i_eq)

end
end
theory Lazy_Greedy_Stateful_Approx
  imports
    Greedy_Submodular_Approx
    Lazy_Greedy_Stateful_StepSpec
begin
```

Approximation guarantee for the verified stateful lazy greedy construction.

This theory treats the stateful lazy algorithm as an implementation-oriented variant of the greedy construction. It reuses the optimal-value infrastructure from the greedy approximation development, together with the per-iteration lemmas from the lazy step-spec theory, and proves a corresponding gap recurrence for the lazy construction.

In particular, this theory does not instantiate the stateless step-spec locale. Instead, it works directly with the verified lazy run and its sequence-level properties.

```
context Cardinality_Constraint
begin

definition gapL :: "nat ⇒ real" where
  "gapL i = OPT_k - f (lazy_set i)"

lemma lazy_set_0[simp]: "lazy_set 0 = {}"
  by (simp add: lazy_set_def init_state_def)

lemma lazy_set_subset_V[simp]: "lazy_set i ⊆ V"
  using lazy_state_subset_V[of i]
  by (simp add: lazy_set_def)
```

```
lemma lazy_set_finite[simp]: "finite (lazy_set i)"
  using finite_V lazy_set_subset_V
  by (meson finite_subset)

lemma remaining_lazy_state[simp]: "remaining (lazy_state i) = V - lazy_set
i"
  by (simp add: remaining_def lazy_set_def)

lemma card_lazy_le_i: "card (lazy_set i) ≤ i"
proof (induction i)
  case 0
  then show ?case by simp
next
  case (Suc i)
  show ?case
  proof (cases "V - lazy_set i = {}")
    case True
    have "lazy_step (lazy_state i) = lazy_state i"
      using lazy_step_idle[of "lazy_state i"] True by simp
    hence "lazy_set (Suc i) = lazy_set i"
      by (simp add: lazy_set_def)
    thus ?thesis using Suc.IH by simp
  next
    case False
    have ins: "lazy_set (Suc i) = insert (lazy_choice i) (lazy_set i)"
      using lazy_set_Suc_insert_V_minus_S[OF False] .
    have xin: "lazy_choice i ∈ V - lazy_set i"
      using lazy_choice_in_V_minus_S[OF False] .
    have xnot: "lazy_choice i ∉ lazy_set i" using xin by simp
    have "card (lazy_set (Suc i)) = card (lazy_set i) + 1"
      using ins xnot by simp
    thus ?thesis using Suc.IH by simp
  qed
qed

lemma card_lazy_lt_k:
  "i < k ⟹ card (lazy_set i) < k"
  using card_lazy_le_i by (meson le_less_trans)

lemma lazy_remainder_nonempty:
  "i < k ⟹ V - lazy_set i ≠ {}"
proof -
  assume i_lt_k: "i < k"

  have "card (lazy_set i) ≤ i"
    by (rule card_lazy_le_i)
  also have "... < k"
    using i_lt_k by simp
  also have "... ≤ card V"
```

```
    using k_le_cardV by simp
  finally have ltV: "card (lazy_set i) < card V" .

  have S_sub: "lazy_set i ⊆ V"
    by simp

  show "V - lazy_set i ≠ {}"
  proof
    assume empty: "V - lazy_set i = {}"
    have V_sub: "V ⊆ lazy_set i"
      using empty by auto
    have eq: "lazy_set i = V"
      using subset_antisym[OF S_sub V_sub] by simp
    thus False
      using ltV by simp
  qed
qed

lemma lazy_set_feasible:
  assumes "i ≤ k"
  shows "feasible (lazy_set i)"
proof -
  have sub: "lazy_set i ⊆ V" by simp
  have card_le_k: "card (lazy_set i) ≤ k"
    using card_lazy_le_i assms by (rule le_trans)
  show ?thesis
    using sub card_le_k
    by (simp add: feasible_def)
qed

lemma gapL_nonneg:
  assumes "i ≤ k"
  shows "0 ≤ gapL i"
proof -
  have feas: "feasible (lazy_set i)"
    using lazy_set_feasible[OF assms] .
  have ub: "f (lazy_set i) ≤ OPT_k"
    by (rule OPT_k_upper_bound[OF feas])
  show ?thesis
    using ub by (simp add: gapL_def)
qed

lemma lazy_step_ineq:
  "i < k ⟹ gain (lazy_set i) (lazy_choice i) ≥ gapL i / real k"
proof -
  assume i_lt_k: "i < k"

  have S_sub: "lazy_set i ⊆ V"
    by simp
```

```
  have cardS_lt_k: "card (lazy_set i) < k"
    using card_lazy_lt_k[OF i_lt_k] .

  obtain X where X_feas: "feasible X" and X_opt: "f X = OPT_k"
    using exists_opt_set by blast

  from X_feas have X_sub: "X ⊆ V" and cardX_le_k: "card X ≤ k"
    unfolding feasible_def by auto

  from marginal_gain_lower_bound[OF S_sub X_sub cardS_lt_k cardX_le_k]
  obtain e where e_in: "e ∈ V - lazy_set i"
      and e_lb: "gain (lazy_set i) e ≥ (f X - f (lazy_set i)) / real
k"
    by blast

  have rem_ne: "V - lazy_set i ≠ {}"
    using lazy_remainder_nonempty[OF i_lt_k] .

  have argmax:
    "∀y∈V - lazy_set i. gain (lazy_set i) y ≤ gain (lazy_set i) (lazy_choice
i)"
    using lazy_choice_argmax_V_minus_S[OF rem_ne] .

  have e_le: "gain (lazy_set i) e ≤ gain (lazy_set i) (lazy_choice i)"
    using argmax e_in by auto

  have e_lb': "gapL i / real k ≤ gain (lazy_set i) e"
    using e_lb X_opt
    by (simp add: gapL_def)

  have "gapL i / real k ≤ gain (lazy_set i) (lazy_choice i)"
    using order_trans[OF e_lb' e_le] .

  thus "gain (lazy_set i) (lazy_choice i) ≥ gapL i / real k"
    by simp
qed

lemma gapL_step:
  "i < k ⟹ gapL (Suc i) ≤ (1 - 1 / real k) * gapL i"
proof -
  assume i_lt_k: "i < k"

  have rem_ne: "V - lazy_set i ≠ {}"
    using lazy_remainder_nonempty[OF i_lt_k] .

  have ins: "lazy_set (Suc i) = insert (lazy_choice i) (lazy_set i)"
    using lazy_set_Suc_insert_V_minus_S[OF rem_ne] .

  have step_gain:
```

```
    "f (lazy_set (Suc i)) = f (lazy_set i) + gain (lazy_set i) (lazy_choice
i)"
    using ins by (simp add: gain_def algebra_simps)

  have gap_suc: "gapL (Suc i) = gapL i - gain (lazy_set i) (lazy_choice
i)"
    by (simp add: gapL_def step_gain)

  have gain_lb: "gain (lazy_set i) (lazy_choice i) ≥ gapL i / real k"
    using lazy_step_ineq[OF i_lt_k] .

  have "gapL (Suc i) ≤ gapL i - gapL i / real k"
    using gap_suc gain_lb by linarith
  also have "... = (1 - 1 / real k) * gapL i"
    by (simp add: algebra_simps)
  finally show "gapL (Suc i) ≤ (1 - 1 / real k) * gapL i" .
qed

lemma gapL_geometric:
  "k > 0 ⟹ i ≤ k ⟹ gapL i ≤ (1 - 1 / real k) ^ i * OPT_k"
proof (induction i)
  case 0
  then show ?case
    by (simp add: gapL_def f_empty)
next
  case (Suc i)
  have i_le_k: "i ≤ k"
    using Suc.prems by simp
  have i_lt_k: "i < k"
    using Suc.prems by simp

  have step: "gapL (Suc i) ≤ (1 - 1 / real k) * gapL i"
    using gapL_step[OF i_lt_k] .

  have coef_nonneg: "0 ≤ (1 - 1 / real k)"
  proof -
    have "1 ≤ real k"
      using Suc.prems(1) by simp
    then have "1 / real k ≤ 1"
      by (simp add: field_simps)
    thus ?thesis
      by simp
  qed

  have IH: "gapL i ≤ (1 - 1 / real k) ^ i * OPT_k"
    using Suc.IH[OF Suc.prems(1) i_le_k] .

  have mult_mono:
    "(1 - 1 / real k) * gapL i
```

```
    ≤ (1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)"
    using IH coef_nonneg
    by (rule mult_left_mono)

  have pow_Suc:
    "(1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)
     = (1 - 1 / real k) ^ Suc i * OPT_k"
    by (simp add: mult_ac)

  have "gapL (Suc i) ≤ (1 - 1 / real k) * ((1 - 1 / real k) ^ i * OPT_k)"
    using step mult_mono
    by (rule order_trans)
  thus ?case
    by (simp add: pow_Suc)
qed

theorem lazy_stateful_approximation:
  assumes k_pos: "k > 0"
  shows "f (lazy_set k) ≥ (1 - 1 / exp 1) * OPT_k"
proof -
  have gap_bound: "gapL k ≤ (1 - 1 / real k) ^ k * OPT_k"
    using gapL_geometric[OF k_pos, of k]
    by simp

  have f_eq: "f (lazy_set k) = OPT_k - gapL k"
    by (simp add: gapL_def)

  have lower: "OPT_k - gapL k ≥ OPT_k - (1 - 1 / real k) ^ k * OPT_k"
    using gap_bound by linarith

  have base_bound: "f (lazy_set k) ≥ (1 - (1 - 1 / real k) ^ k) * OPT_k"
  proof -
    have "f (lazy_set k) ≥ OPT_k - (1 - 1 / real k) ^ k * OPT_k"
      using f_eq lower by simp
    also have "OPT_k - (1 - 1 / real k) ^ k * OPT_k
               = (1 - (1 - 1 / real k) ^ k) * OPT_k"
      by (simp add: algebra_simps)
    finally show ?thesis .
  qed

  have k_ge1: "k ≥ 1"
    using k_pos by simp

  have coeff_bound: "1 - (1 - 1 / real k) ^ k ≥ 1 - 1 / exp 1"
    using coeff_ge_1_minus_inv_exp[OF k_ge1] .

  have nonneg: "0 ≤ OPT_k"
    by (rule OPT_k_nonneg)
```

```
  have coeff_mono:
    "(1 - (1 - 1 / real k) ^ k) * OPT_k ≥ (1 - 1 / exp 1) * OPT_k"
    using coeff_bound nonneg
    by (rule mult_right_mono)

  show "f (lazy_set k) ≥ (1 - 1 / exp 1) * OPT_k"
    using base_bound coeff_mono
    by (meson order_trans)
qed

end
```

6 Acknowledgements

The author is grateful to Wenda Li for careful reviews, comments, and guidance from the early stages of this project through the preparation of this AFP entry.

end

References

[1] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. In *Optimization Techniques*, pages 234–243. Springer, 1978.

[2] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions—I. *Mathematical Programming*, 14(1):265–294, 1978.