Submodularity Practical

Matthew Staib, Stefanie Jegelka

Outline

- Motivating application: summarization
- Algorithms:
  - Monotone constrained
  - Non-monotone unconstrained
- Return to summarization

Application: summarization

Application: summarization

- Super large set *V* of items, e.g. images, documents

Application: summarization

- Super large set *V* of items, e.g. images, documents
- Goal: find small subset *S* that can represent *V*

Application: summarization

- Super large set *V* of items, e.g. images, documents
- Goal: find small subset *S* that can represent *V*
- Idea: define set function $f(S)$ capturing quality of *S*

Application: summarization

- Super large set *V* of items, e.g. images, documents
- Goal: find small subset *S* that can represent *V*
- Idea: define set function $f(S)$ capturing quality of *S*
- Then solve $\max_{S:|S|\leq k} f(S)$

Monotone constrained submodular maximization

$$\max_{S:|S|\leq k} f(S)$$

Monotone constrained submodular maximization

$$\max_{S:|S|\leq k} f(S)$$

Can be more general, e.g. matroid constraint

Monotone constrained submodular maximization

$$\max_{S:|S|\leq k} f(S)$$

Can be more general, e.g. matroid constraint

- Greedy

Monotone constrained submodular maximization

$$\max_{S:|S|\leq k} f(S)$$

Can be more general, e.g. matroid constraint

- Greedy
- Making greedy faster:

Monotone constrained submodular maximization

$$\max_{S:|S|\leq k} f(S)$$

Can be more general, e.g. matroid constraint

- Greedy
- Making greedy faster:
  - Lazy acceleration of greedy

Monotone constrained submodular maximization

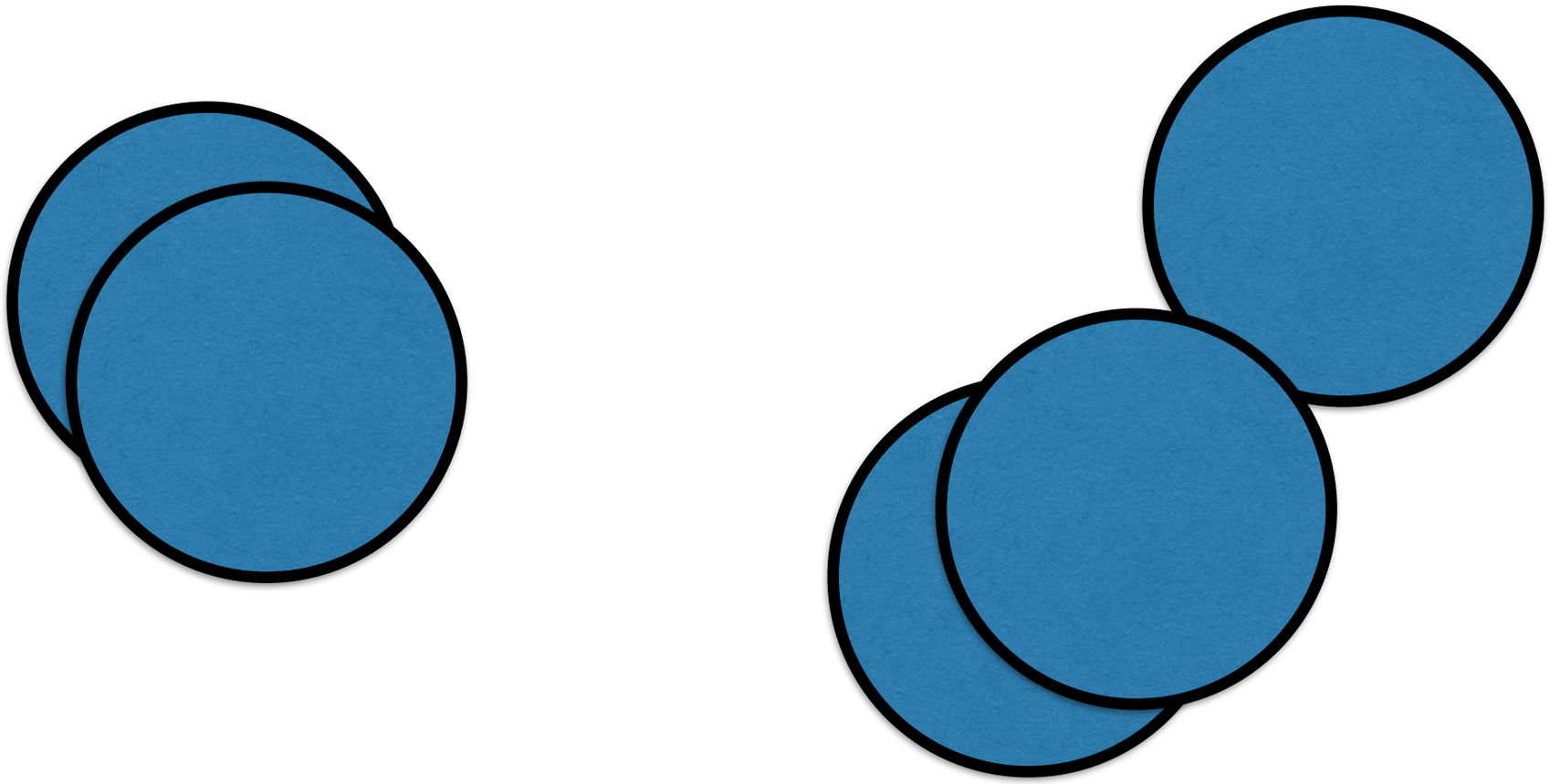
$$\max_{S:|S|\leq k} f(S)$$

Can be more general, e.g. matroid constraint

- Greedy
- Making greedy faster:
  - Lazy acceleration of greedy
  - Stochastic acceleration of greedy

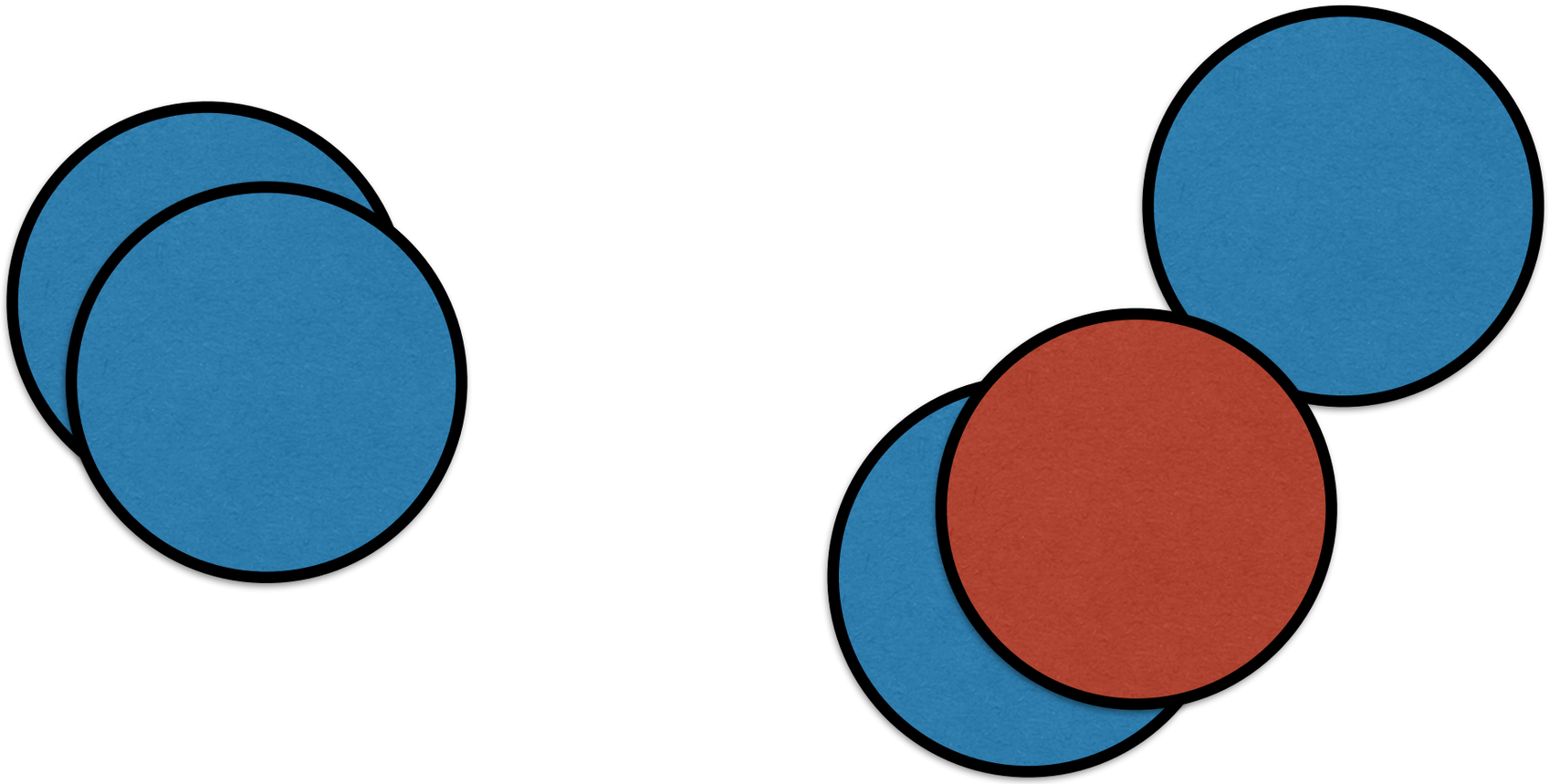
Greedy

Pick k=3 sensors



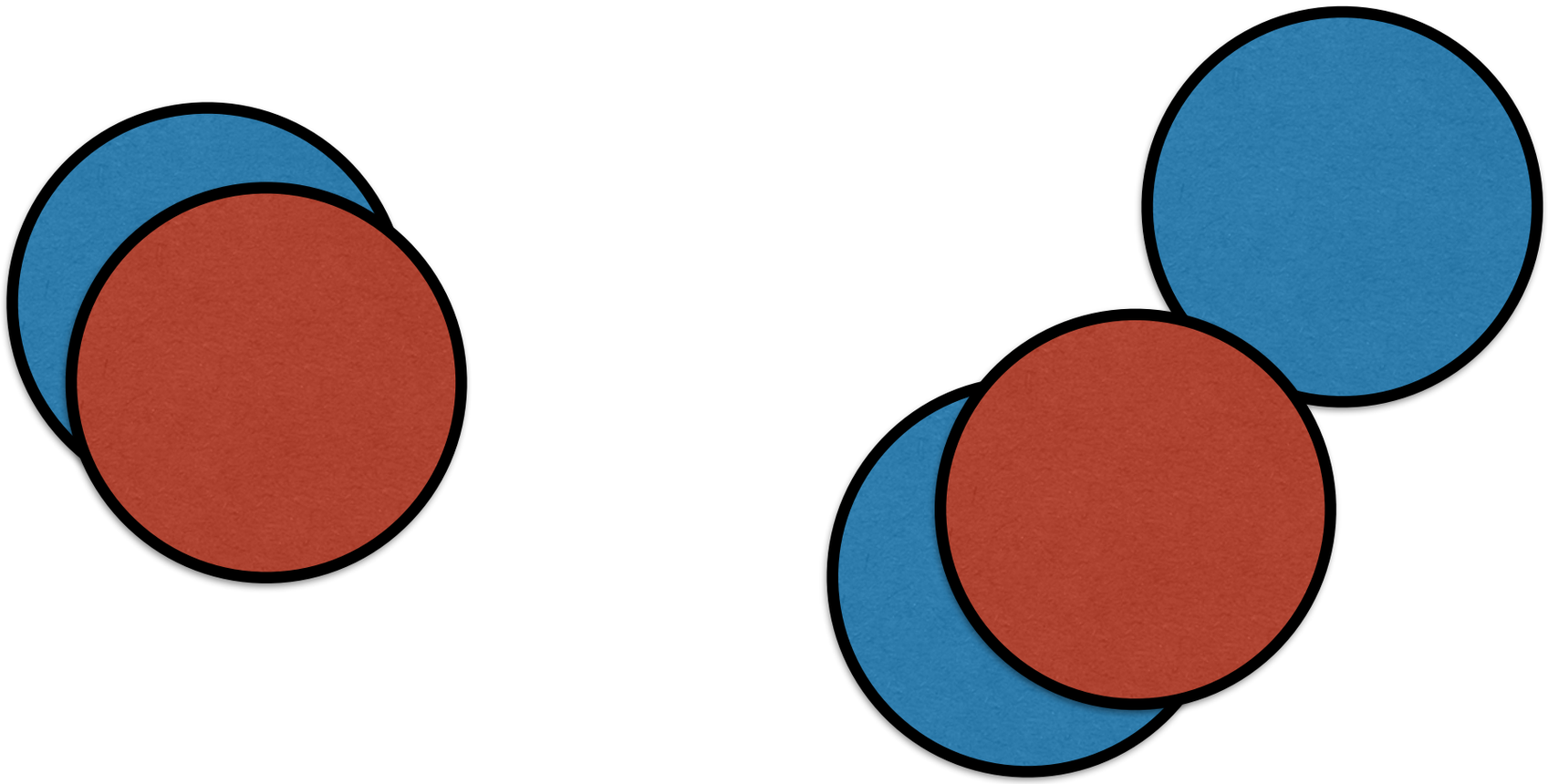
Greedy

Pick k=3 sensors



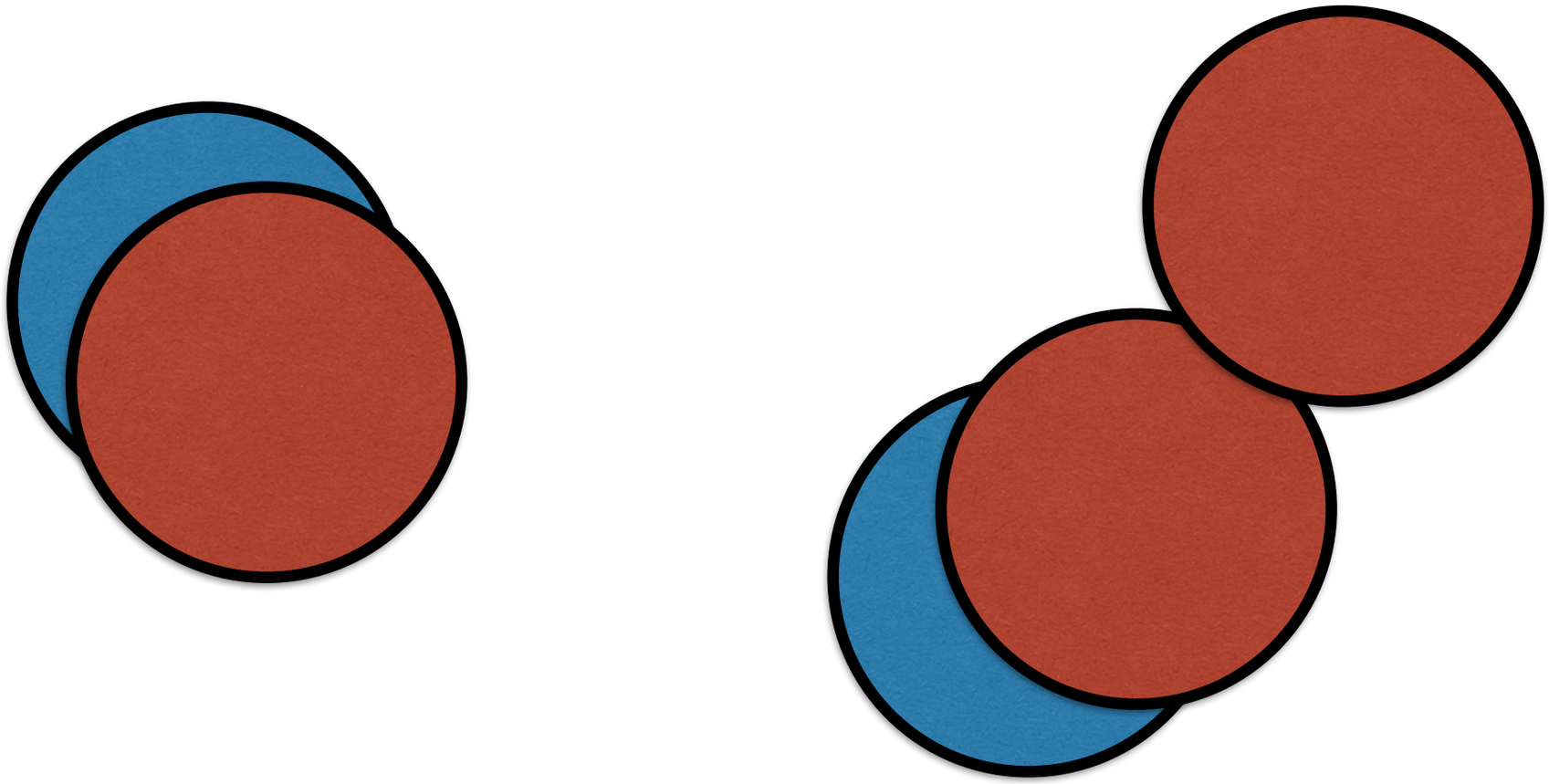
Greedy

Pick k=3 sensors



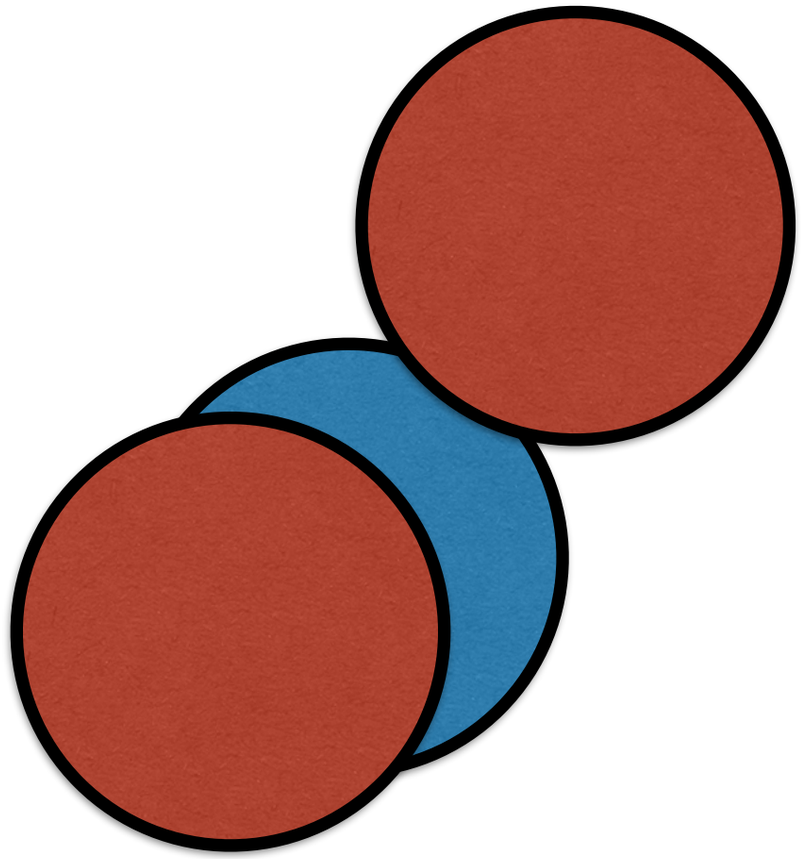
Greedy

Pick k=3 sensors



Greedy

Not guaranteed to be optimal!



Greedy

- Iteratively add item *i* maximizing marginal gains:

$$
\begin{aligned}
&S_0 \leftarrow \emptyset \\
&\textbf{for } t = 0, \ldots, k-1 \textbf{ do} \\
&\quad i^* \leftarrow \arg\max_{i \in V \setminus S} f(S_t \cup \{i\}) \\
&\quad S_{t+1} \leftarrow S_t \cup \{i\} \\
&\textbf{end for} \\
&\textbf{return } S_k
\end{aligned}
$$

Greedy

- Iteratively add item *i* maximizing marginal gains:

$$
\begin{aligned}
&S_0 \leftarrow \emptyset \\
&\textbf{for } t = 0, \ldots, k-1 \textbf{ do} \\
&\quad i^* \leftarrow \arg\max_{i \in V \setminus S} \Delta(S_t, i) \\
&\quad S_{t+1} \leftarrow S_t \cup \{i\} \\
&\textbf{end for} \\
&\textbf{return } S_k
\end{aligned}
$$

$$\Delta(S, i) = f(S \cup \{i\}) - f(S)$$

How can we speed this up?

How can we speed this up?

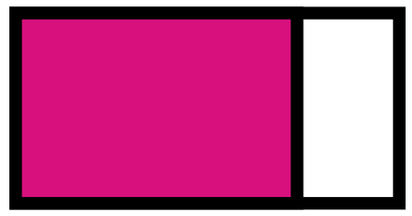
(greedy code: www.mit.edu/~mstaib/mlss/greedy.txt)

Idea 1: leverage submodularity

- At round *t*, pick new element, add to form S_t
- S_t grows each round, i.e. $S_{t-1} \subseteq S_t$
- By submodularity, $\Delta(S_t, i) \leq \Delta(S_{t-1}, i)$

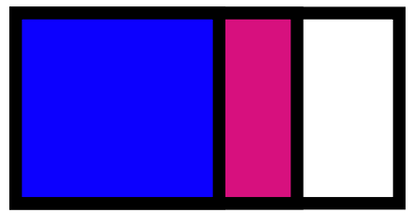
Idea 1: leverage submodularity

- At round *t*, pick new element, add to form S_t
- S_t grows each round, i.e. $S_{t-1} \subseteq S_t$
- By submodularity, $\Delta(S_t, i) \leq \Delta(S_{t-1}, i)$



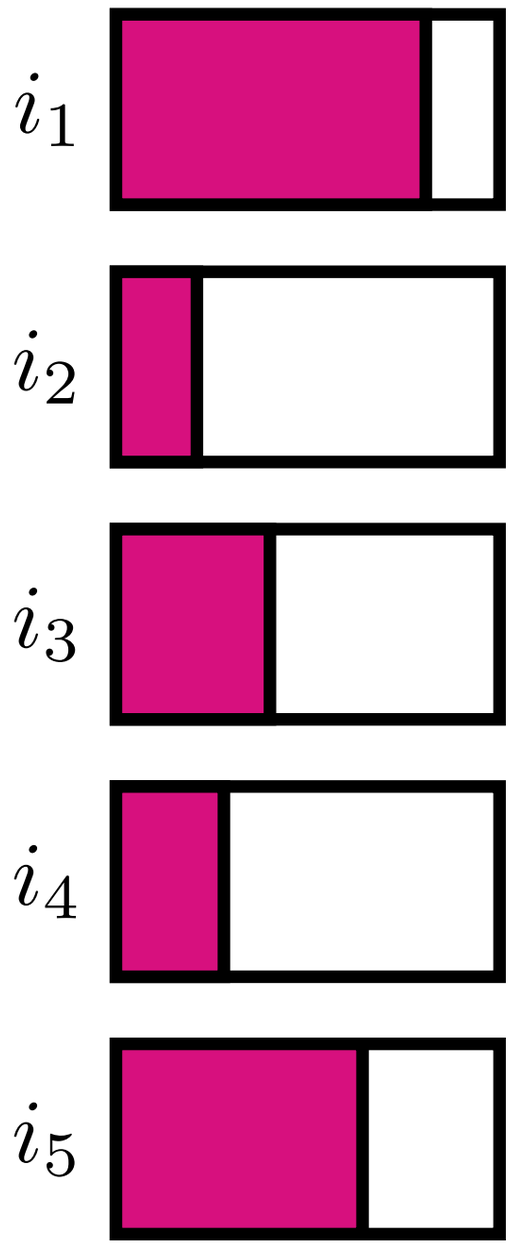
Idea 1: leverage submodularity

- At round *t*, pick new element, add to form S_t
- S_t grows each round, i.e. $S_{t-1} \subseteq S_t$
- By submodularity, $\Delta(S_t, i) \leq \Delta(S_{t-1}, i)$



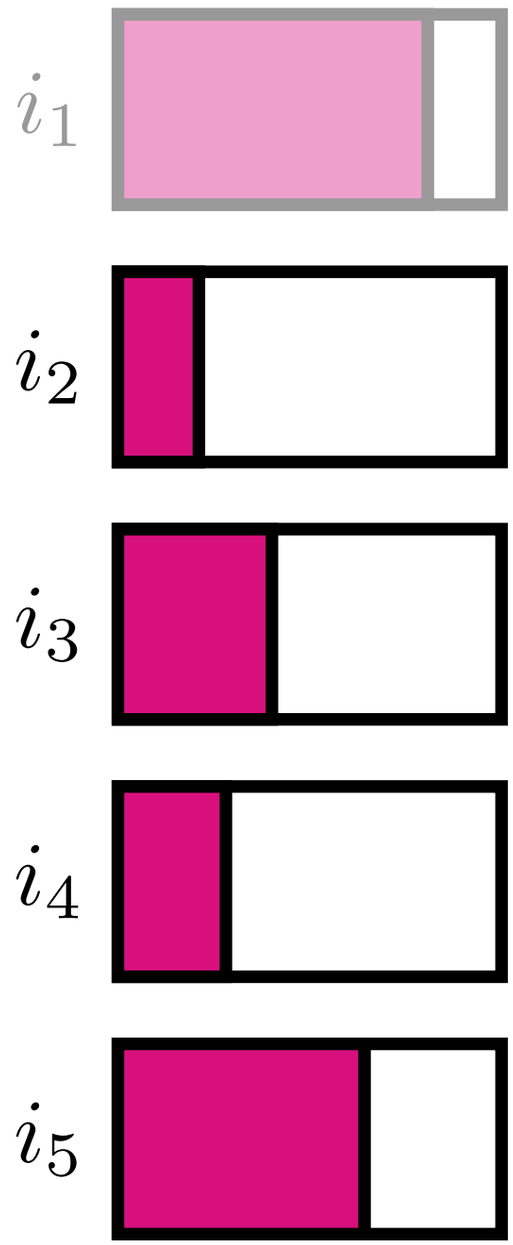
Lazy greedy

- First iteration as usual, take best element



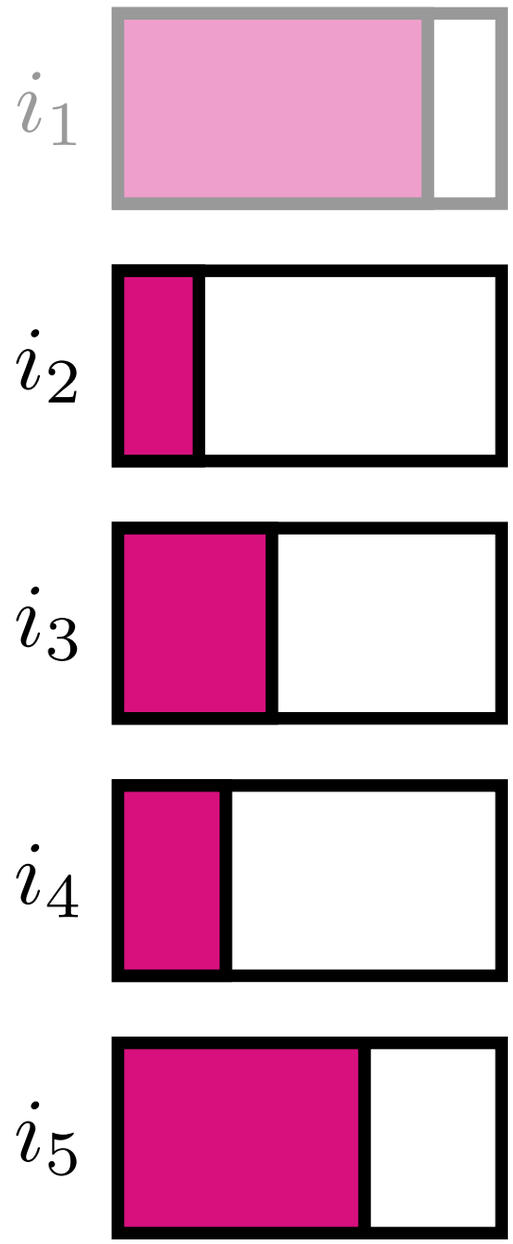
Lazy greedy

- First iteration as usual, take best element



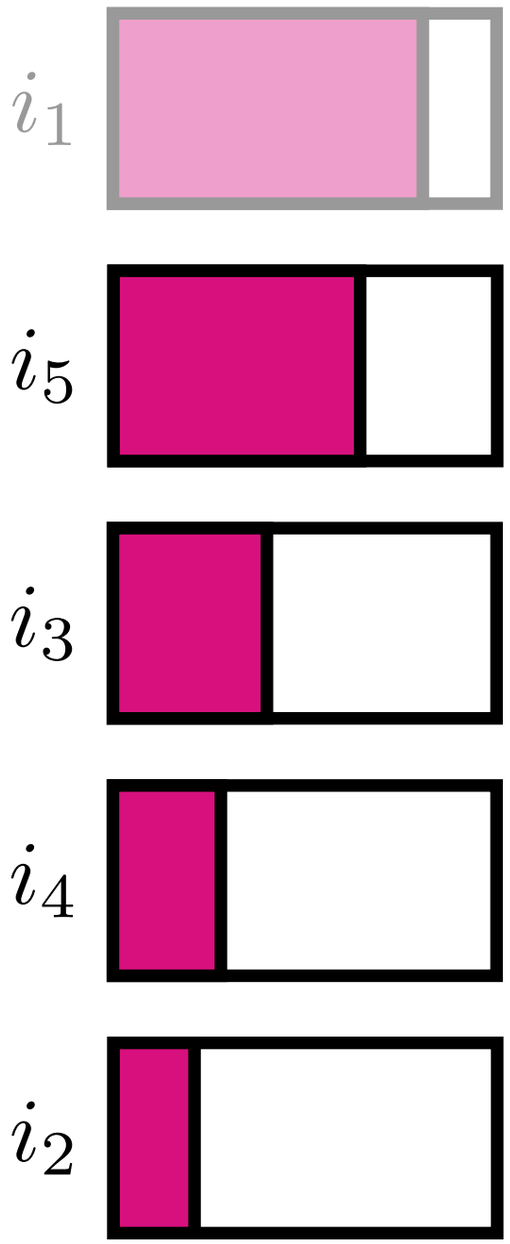
Lazy greedy

- First iteration as usual, take best element
- Maintain <u>sorted list</u> of remaining elements



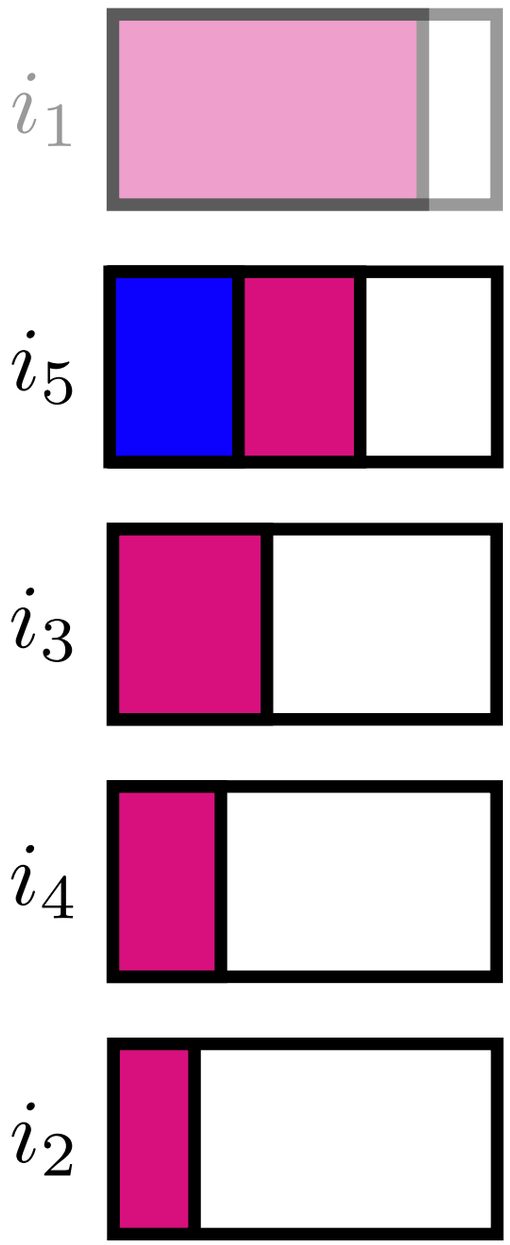
Lazy greedy

- First iteration as usual, take best element
- Maintain <u>sorted list</u> of remaining elements



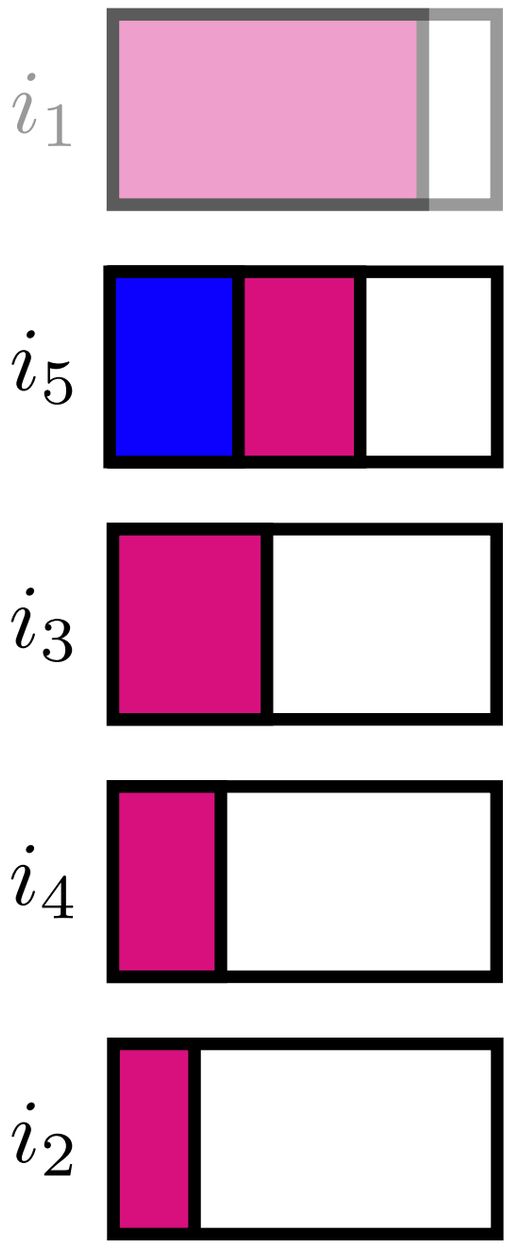
Lazy greedy

- First iteration as usual, take best element
- Maintain <u>sorted list</u> of remaining elements
- Re-compute Δ for top element <u>only</u>



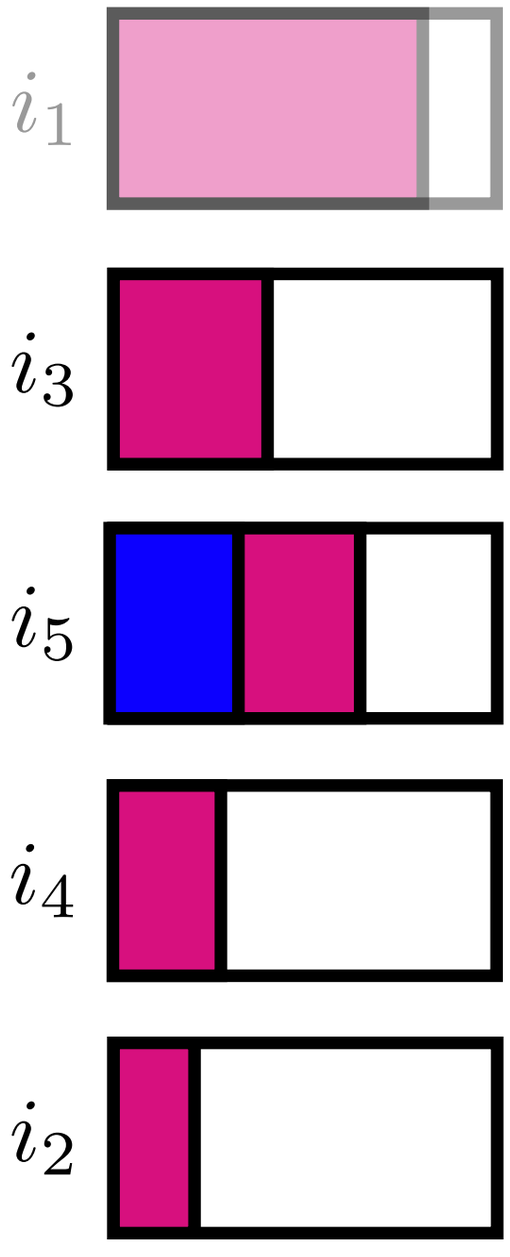
Lazy greedy

- First iteration as usual, take best element
- Maintain sorted list of remaining elements
- Re-compute Δ for top element only
- If still top, take, otherwise re-sort



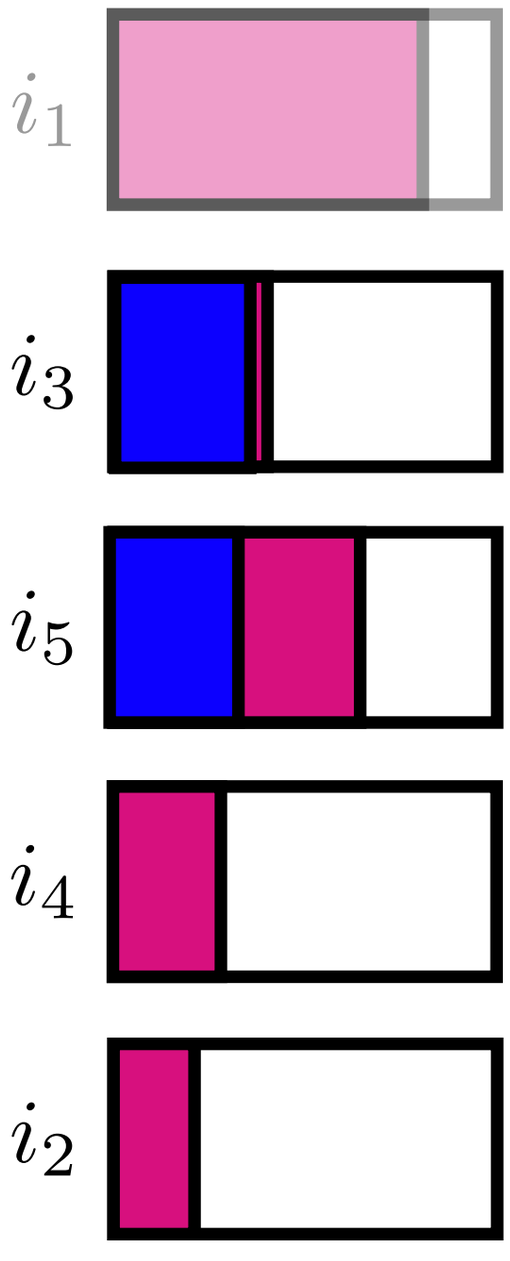
Lazy greedy

- First iteration as usual, take best element
- Maintain sorted list of remaining elements
- Re-compute Δ for top element only
- If still top, take, otherwise re-sort



Lazy greedy

- First iteration as usual, take best element
- Maintain <u>sorted list</u> of remaining elements
- Re-compute Δ for top element <u>only</u>
- If <u>still</u> top, take, otherwise <u>re-sort</u>



Idea 2: subsample

$S_0 \leftarrow \emptyset$
for $t = 0, \ldots, k-1$ **do**
    $i^* \leftarrow \arg\max_{i \in V \setminus S} f(S_t \cup \{i\})$
    $S_{t+1} \leftarrow S_t \cup \{i\}$
end for
return S_k

Idea 2: subsample

$S_0 \leftarrow \emptyset$

for $t = 0, \ldots, k-1$ **do**

$i^* \leftarrow \arg\max_{i \in V \setminus S} f(S_t \cup \{i\})$

$S_{t+1} \leftarrow S_t \cup \{i\}$

end for

return S_k

Idea 2: subsample

$S_0 \leftarrow \emptyset$

for $t = 0, \ldots, k-1$ **do**

$i^* \leftarrow \arg\max_{i \in V \setminus S} f(S_t \cup \{i\})$

$S_{t+1} \leftarrow S_t \cup \{i\}$

end for

return S_k

can be large

Idea 2: subsample

$S_0 \leftarrow \emptyset$

for $t = 0, \ldots, k-1$ **do**

$i^* \leftarrow \arg\max_{i \in V \setminus S} f(S_t \cup \{i\})$

$S_{t+1} \leftarrow S_t \cup \{i\}$

end for

return S_k

can be large

Stochastic greedy:
Each iteration,
- subsample *s* elements
- pick best

Hands-on Section

- Implement stochastic greedy

Hands-on Section

- Implement stochastic greedy
- Compare both to lazy greedy (given)

Hands-on Section

- Implement stochastic greedy
- Compare both to lazy greedy (given)
- When does each algorithm perform well?

Test problems

- Modular functions:

$$f(S) = \sum_{i \in S} w_i$$

Test problems

- Modular functions:

$$f(S) = \sum_{i \in S} w_i$$

- Concave-over-modular: for g concave,

$$f(S) = g\left(\sum_{i \in S} w_i\right)$$

Test problems

- Modular functions:

$$f(S) = \sum_{i \in S} w_i$$

- Concave-over-modular: for *g* concave,

$$f(S) = g\left(\sum_{i \in S} w_i\right)$$

- How does the distribution of weights affect the alg?

Non-monotone unconstrained submodular maximization

$$\max_{S} f(S)$$

Non-monotone unconstrained submodular maximization

$$\max_{S} f(S)$$

No constraints!

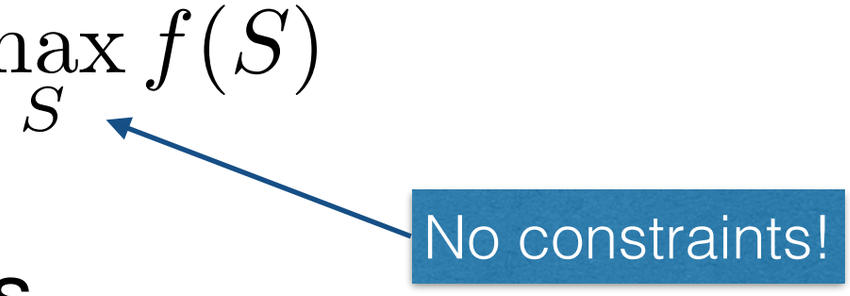
Non-monotone unconstrained submodular maximization

$$\max_S f(S)$$

No constraints!

- Example: cut functions

Non-monotone unconstrained submodular maximization



- Example: cut functions
- Algorithms:

Non-monotone unconstrained submodular maximization

$$\max_S f(S)$$

No constraints!

- Example: cut functions
- Algorithms:
  - Random sampling

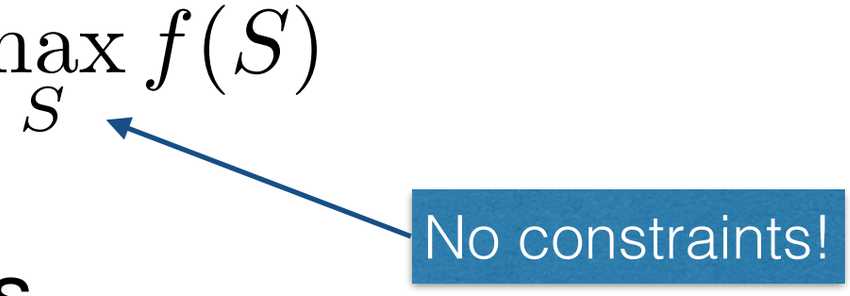
Non-monotone unconstrained submodular maximization

$$\max_S f(S)$$

No constraints!

- Example: cut functions
- Algorithms:
  - Random sampling
  - Deterministic double greedy

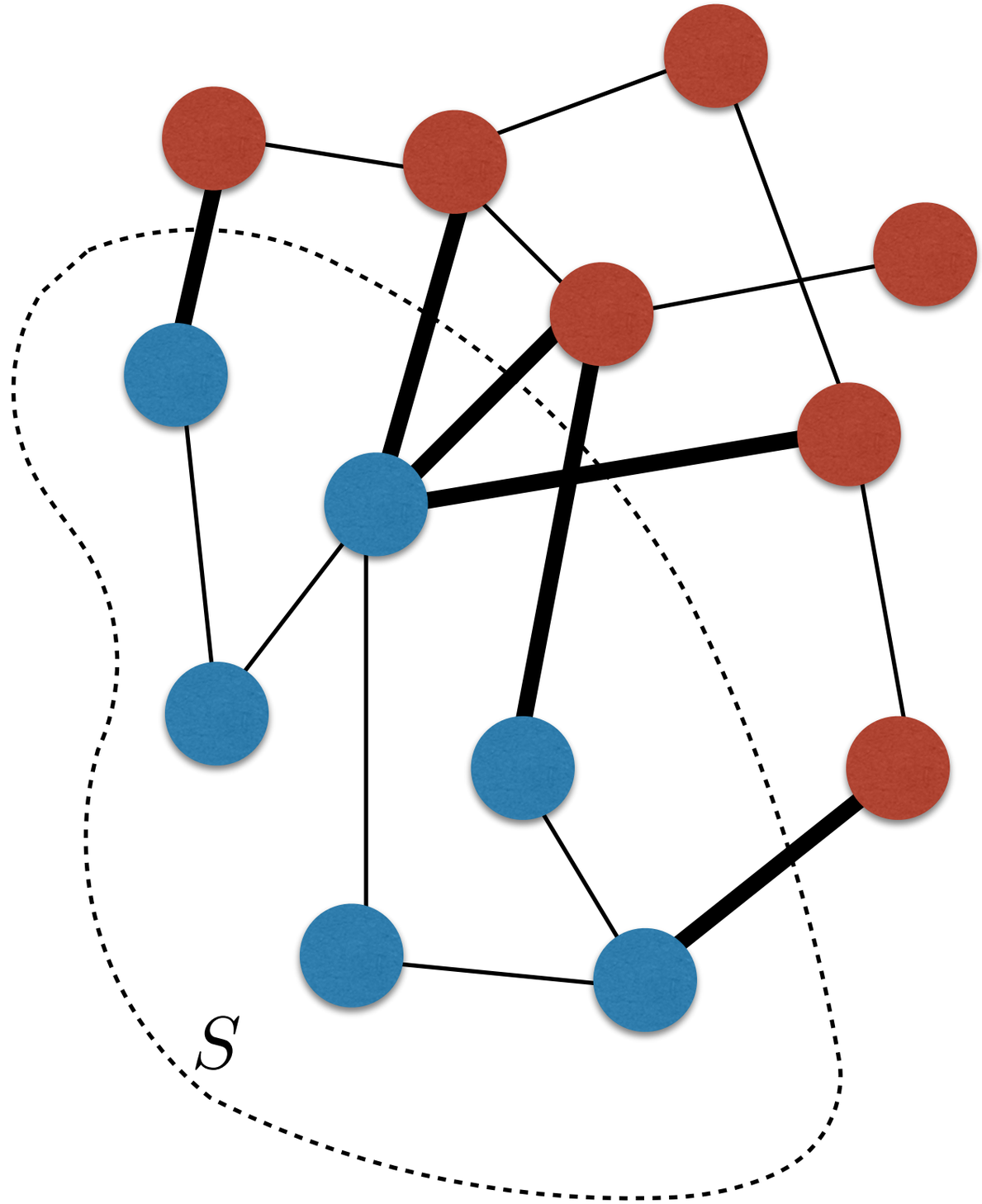
Non-monotone unconstrained submodular maximization



- Example: cut functions
- Algorithms:
  - Random sampling
  - Deterministic double greedy
  - Randomized double greedy

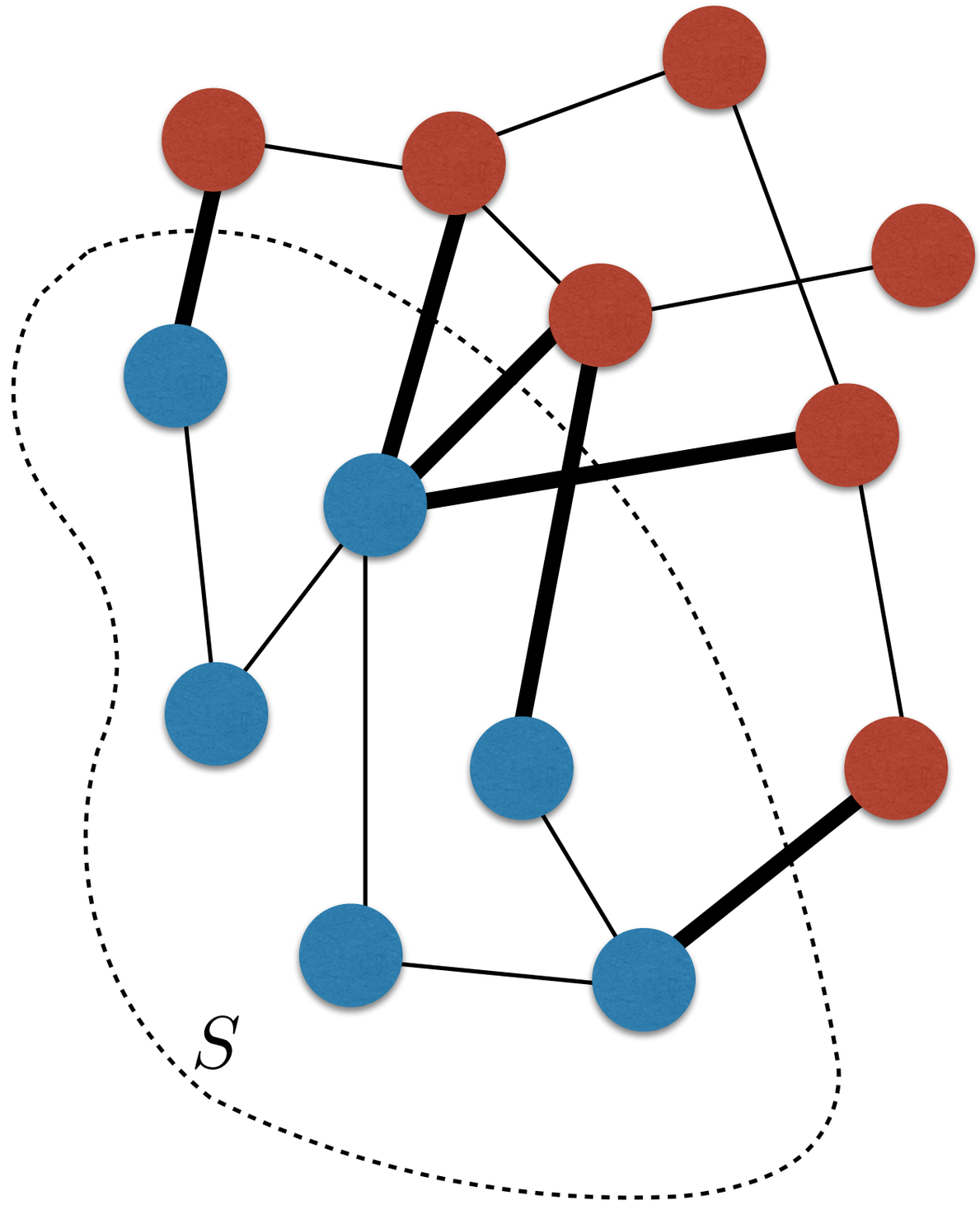
Cut functions

- Graph with edge weights a_{ij}



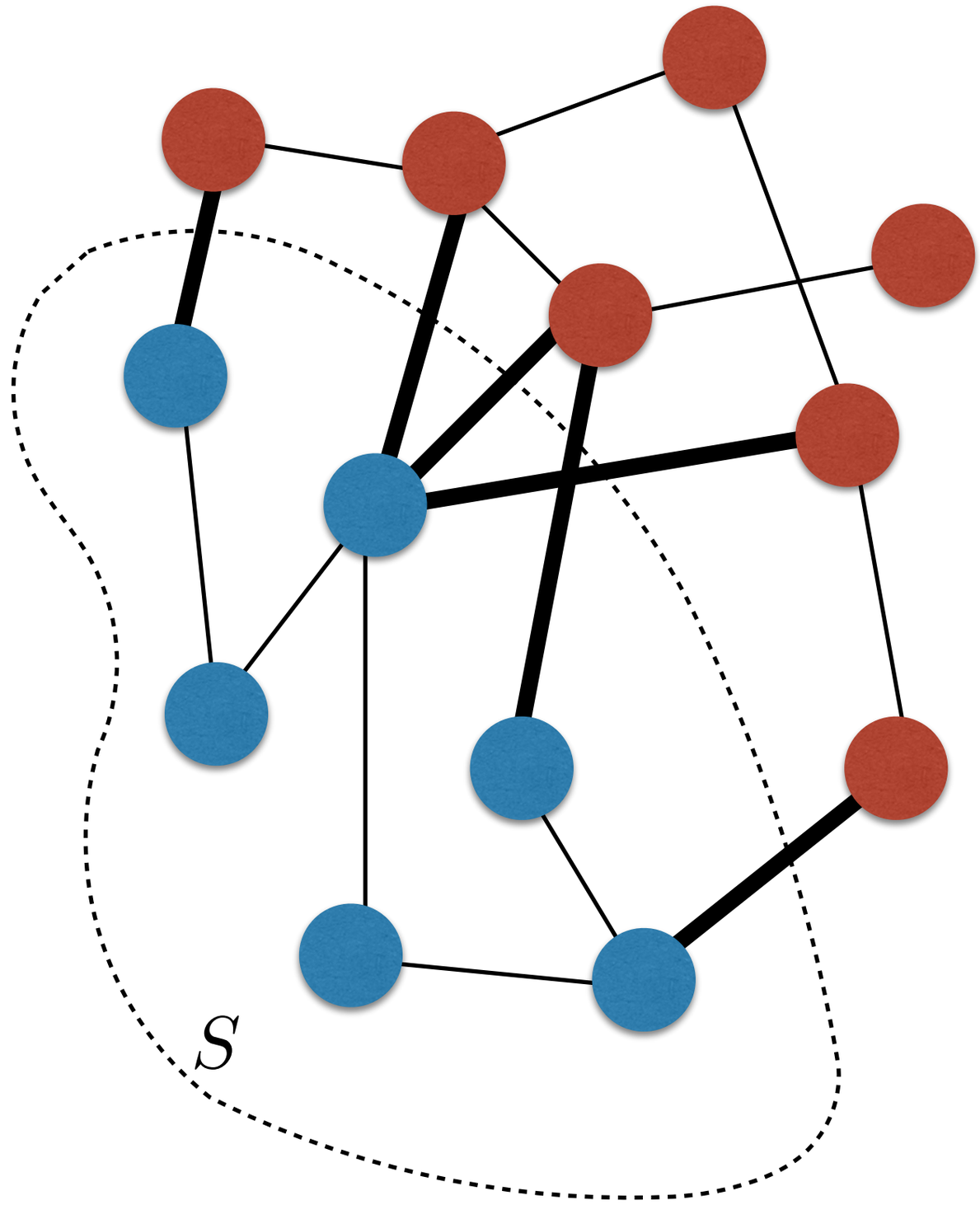
Cut functions

- Graph with edge weights a_{ij}
- $f(S) := \sum_{i \in S} \sum_{j \in V \setminus S} a_{ij}$



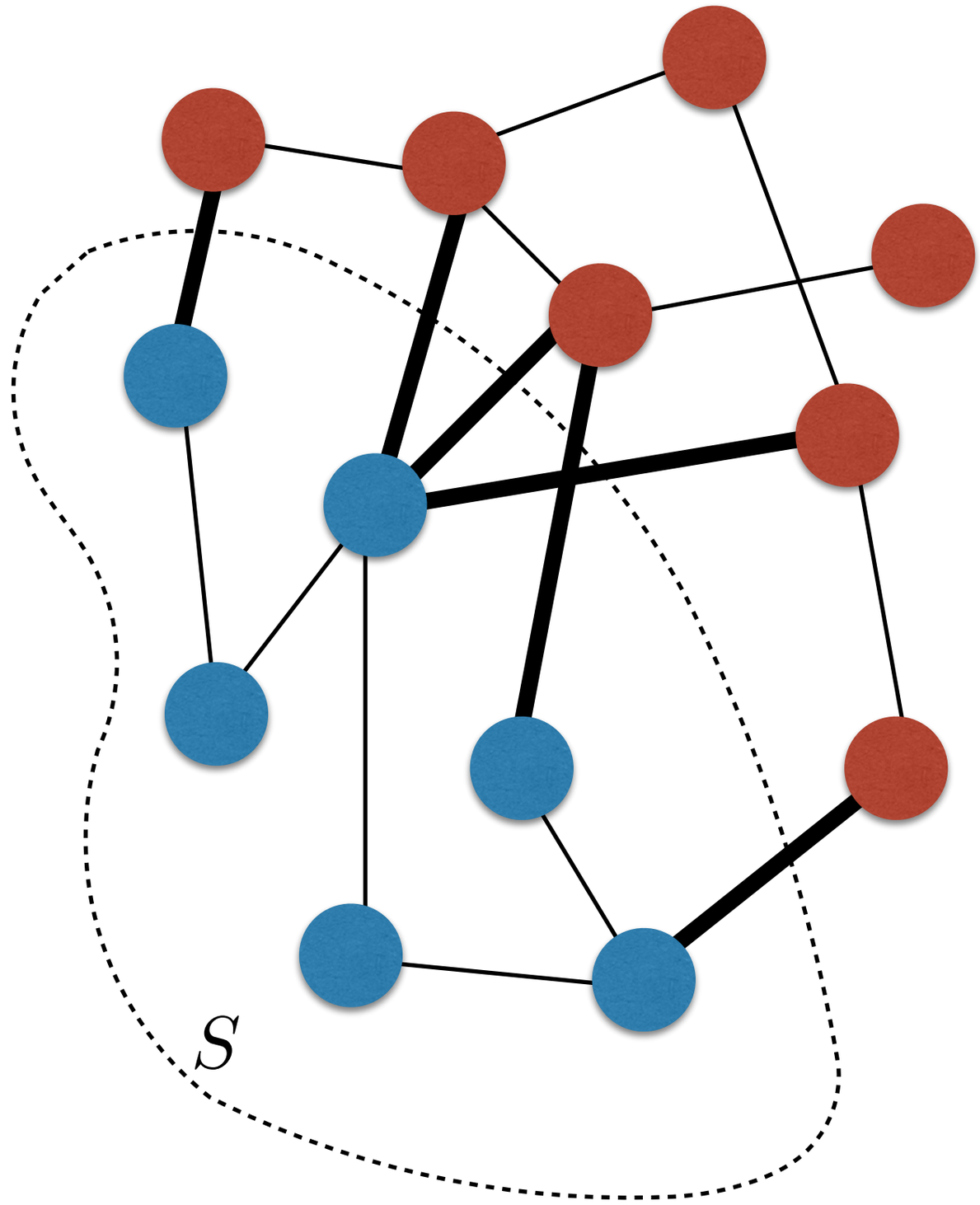
Cut functions

- Graph with edge weights a_{ij}
- $f(S) := \sum_{i \in S} \sum_{j \in V \setminus S} a_{ij}$
- Sum weights across the partition



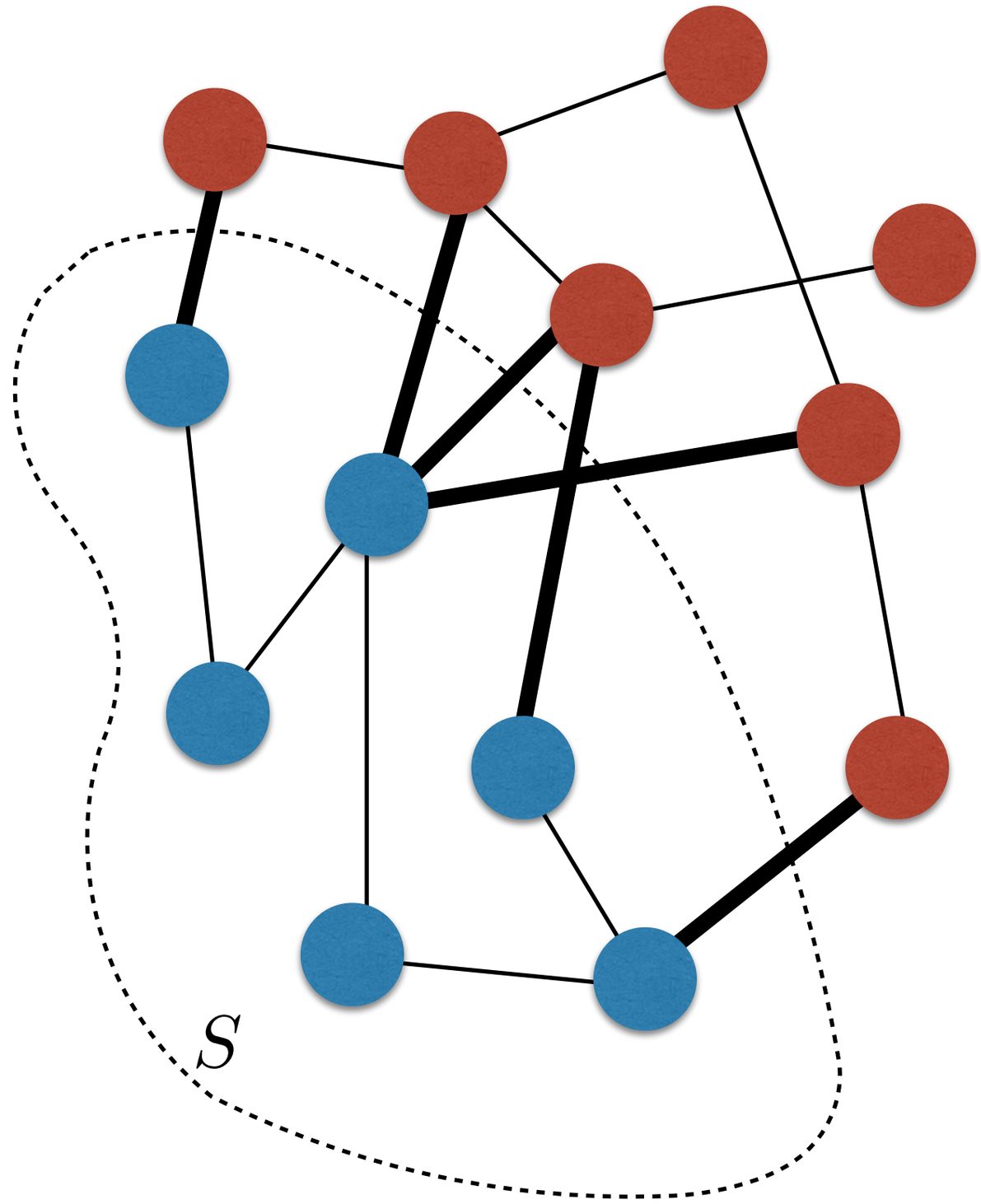
Cut functions

- Graph with edge weights a_{ij}
- $f(S) := \sum_{i \in S} \sum_{j \in V \setminus S} a_{ij}$
- Sum weights across the partition
- Submodular, non-monotone



Cut functions

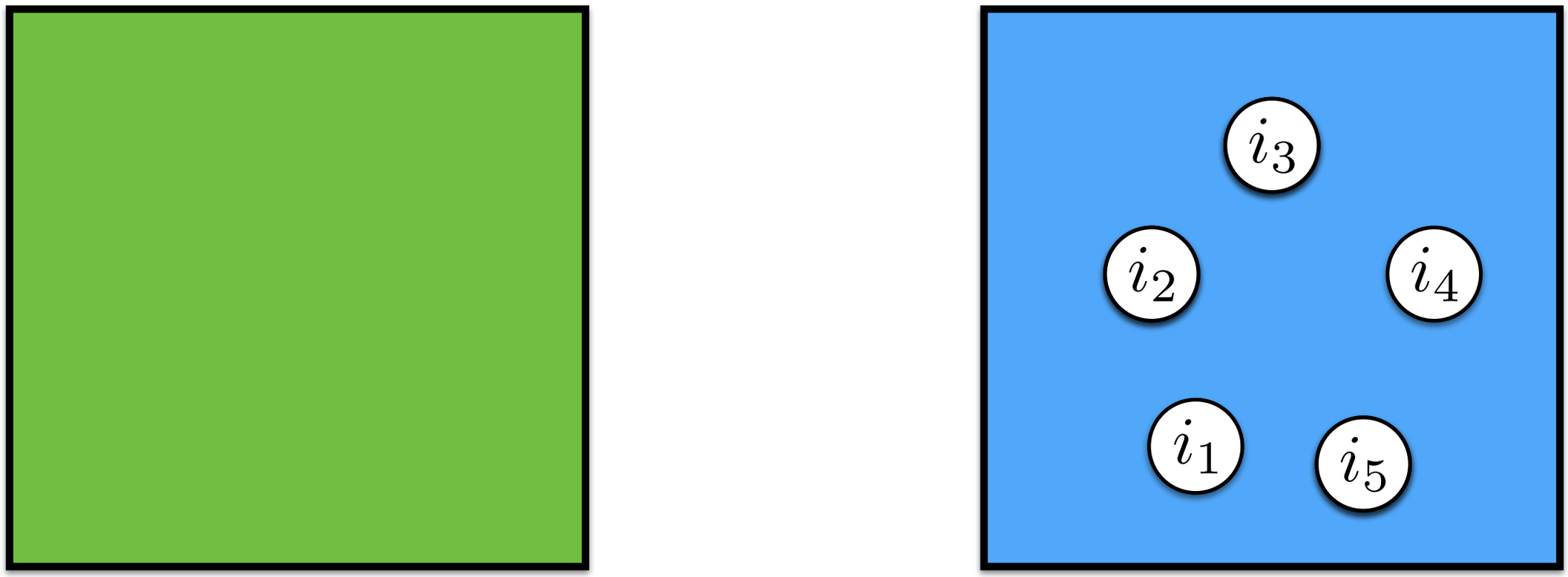
- Graph with edge weights a_{ij}
- $f(S) := \sum_{i \in S} \sum_{j \in V \setminus S} a_{ij}$
- Sum weights across the partition
- Submodular, non-monotone
- MAX CUT



Alg 1: Random sampling

- Choose *S* uniformly at random
- (yes, really)
- Gives a 1/4-approximation

Deterministic double greedy



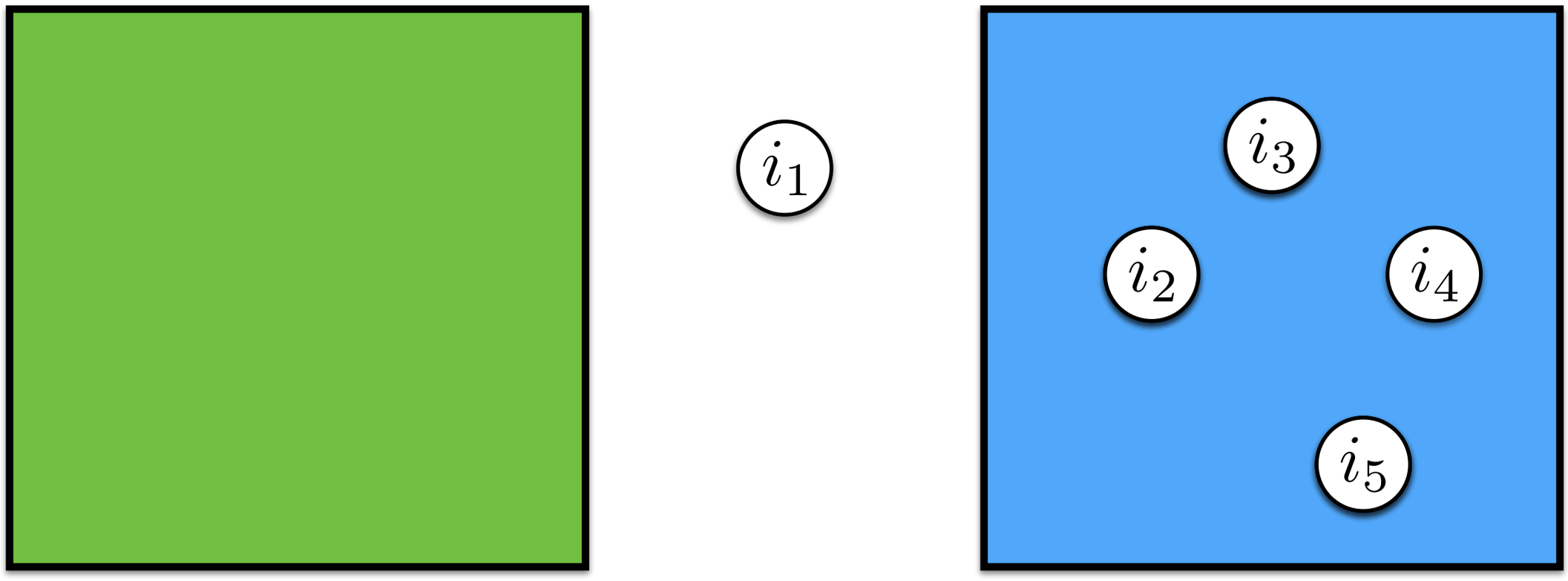
A

B

“definitely keep”

“maybe keep”

Deterministic double greedy



A

"definitely keep"

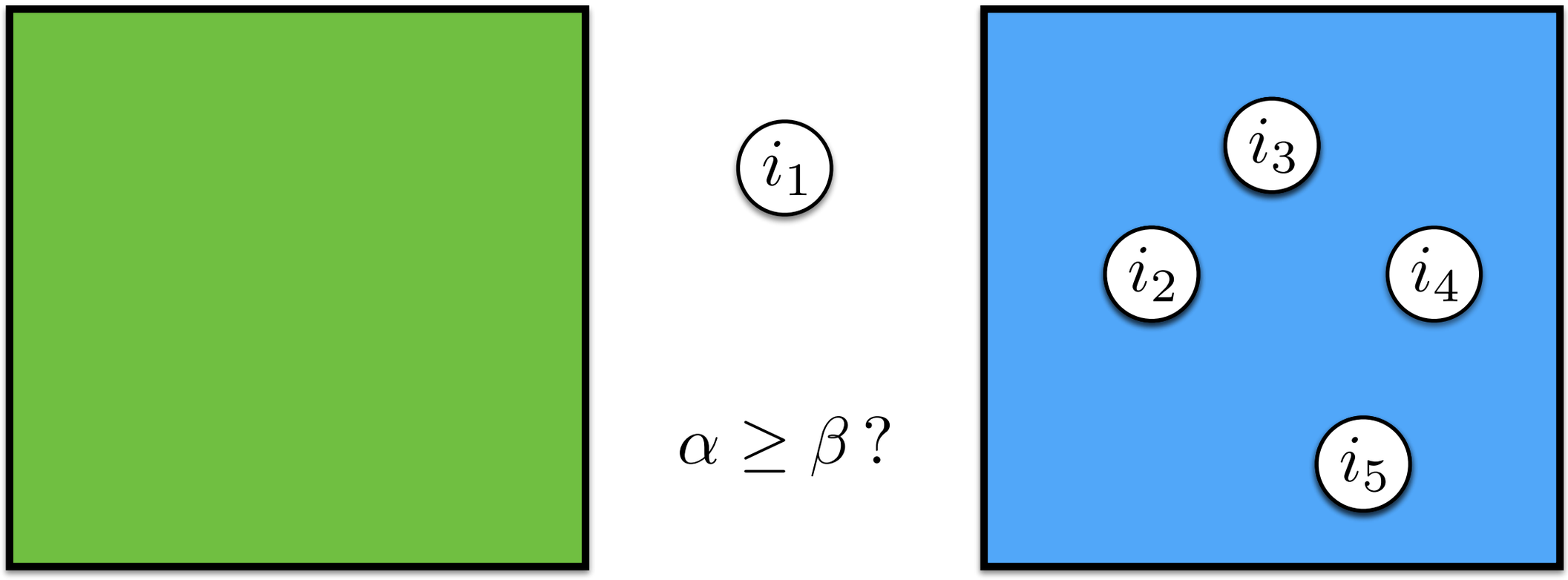
B

"maybe keep"

Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_1\}) - f(A)$ "gain"

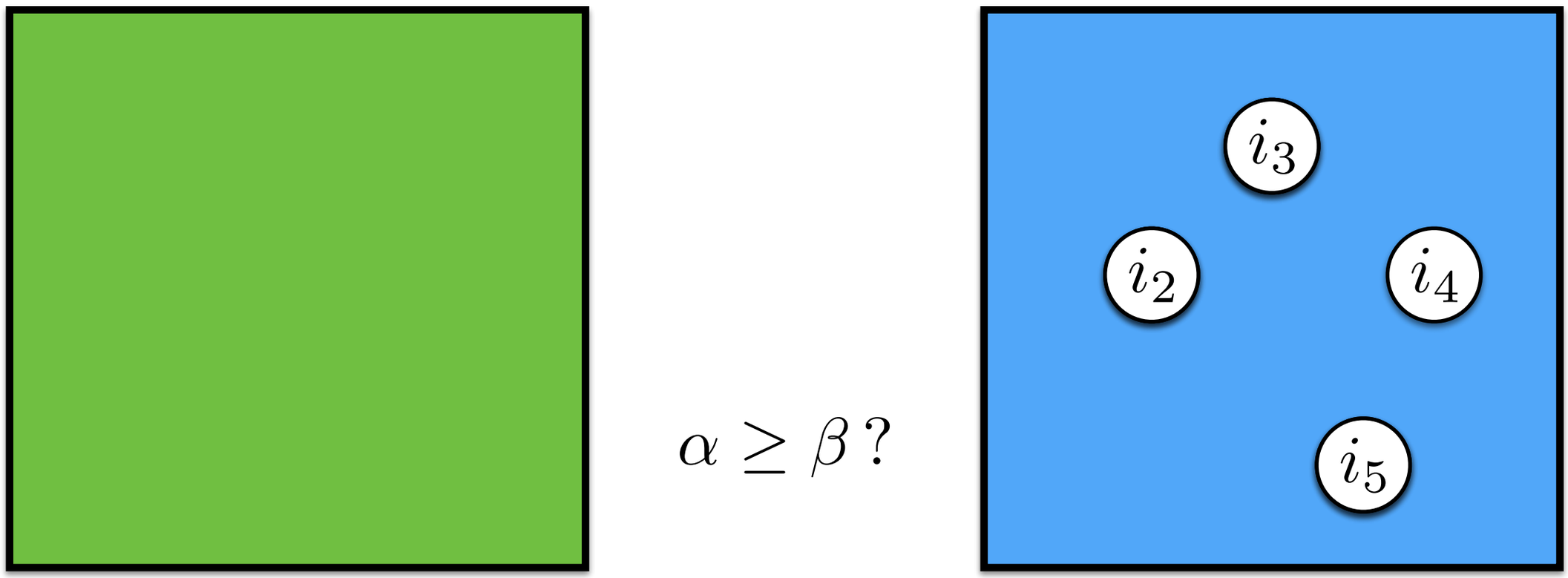
$\beta \leftarrow f(B \setminus \{i_1\}) - f(B)$ "harm"



Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_1\}) - f(A)$ "gain"

$\beta \leftarrow f(B \setminus \{i_1\}) - f(B)$ "harm"



$\alpha \geq \beta\,?$

A

"definitely keep"

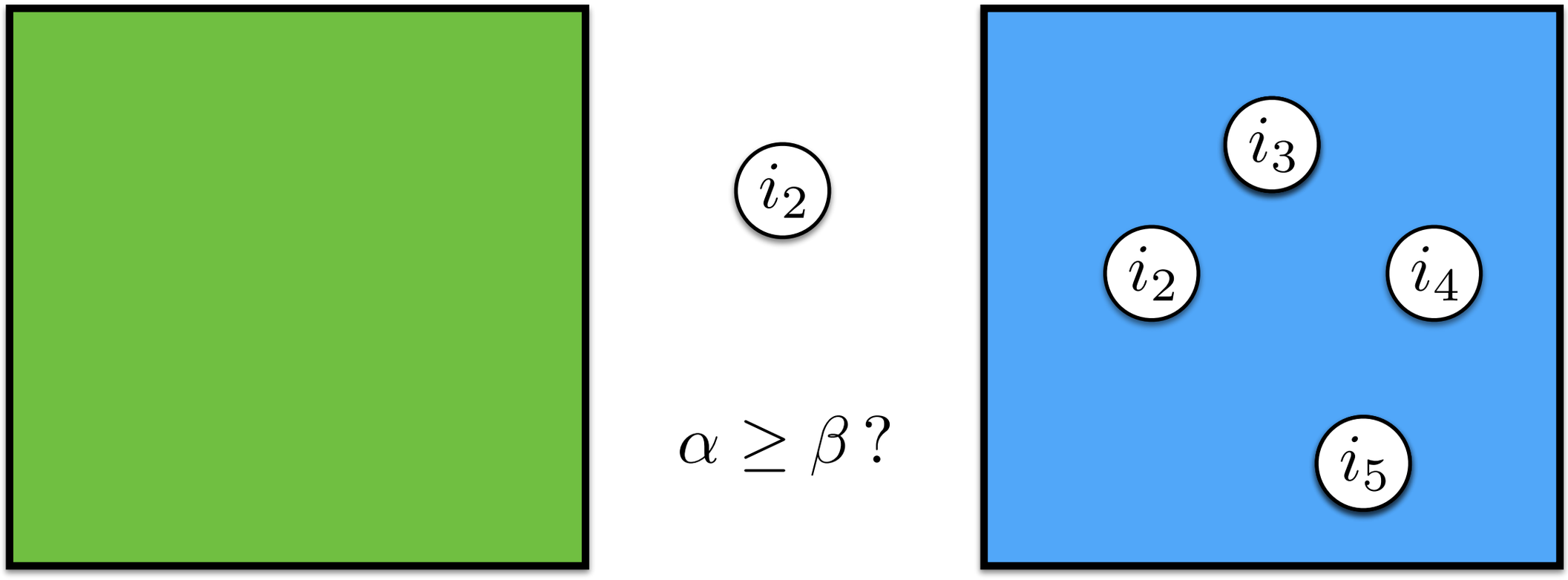
B

"maybe keep"

Deterministic double greedy

$$\alpha \leftarrow f(A \cup \{i_2\}) - f(A) \quad \text{“gain”}$$

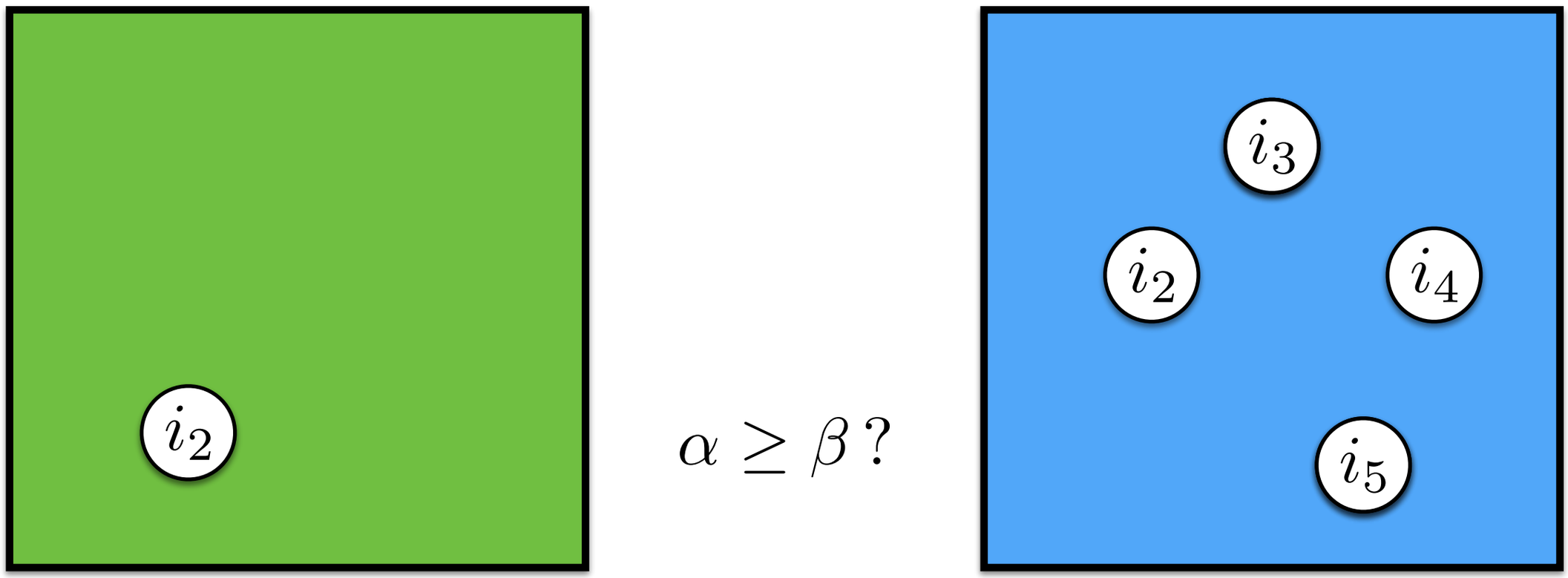
$$\beta \leftarrow f(B \setminus \{i_2\}) - f(B) \quad \text{“harm”}$$



Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_2\}) - f(A)$ "gain"

$\beta \leftarrow f(B \setminus \{i_2\}) - f(B)$ "harm"



$\alpha \geq \beta\,?$

A

"definitely keep"

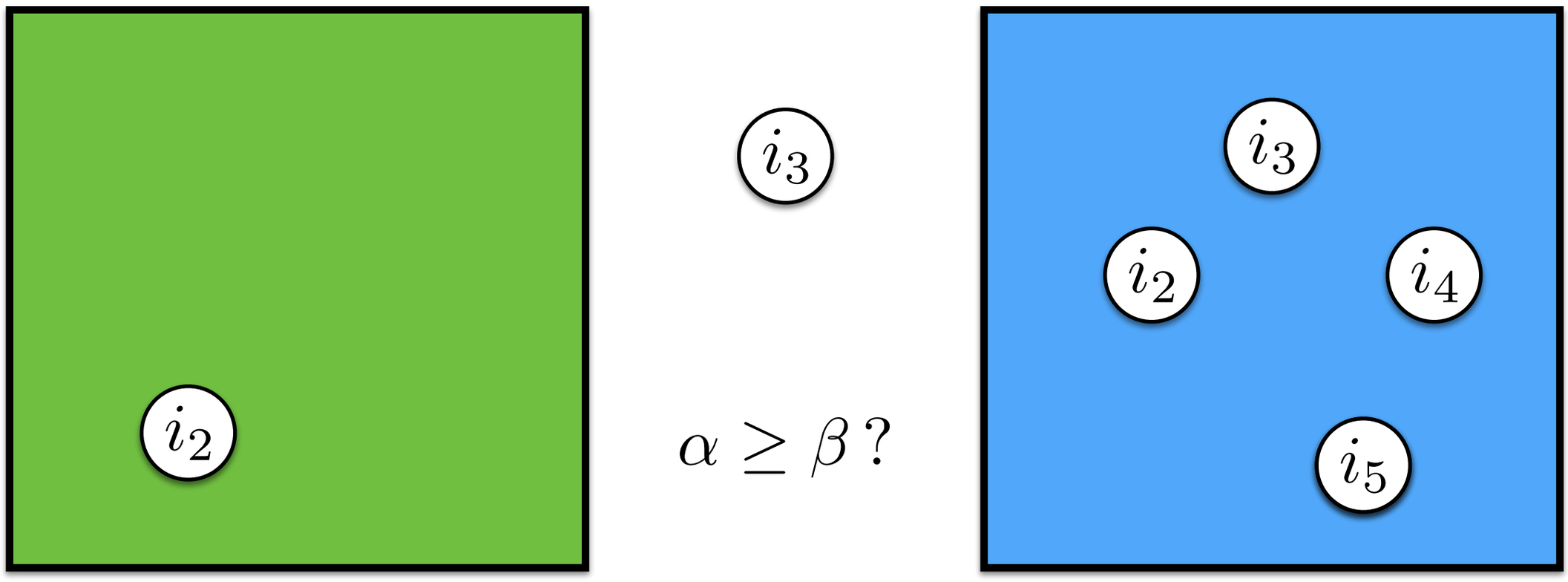
B

"maybe keep"

Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_3\}) - f(A)$ "gain"

$\beta \leftarrow f(B \setminus \{i_3\}) - f(B)$ "harm"



A

"definitely keep"

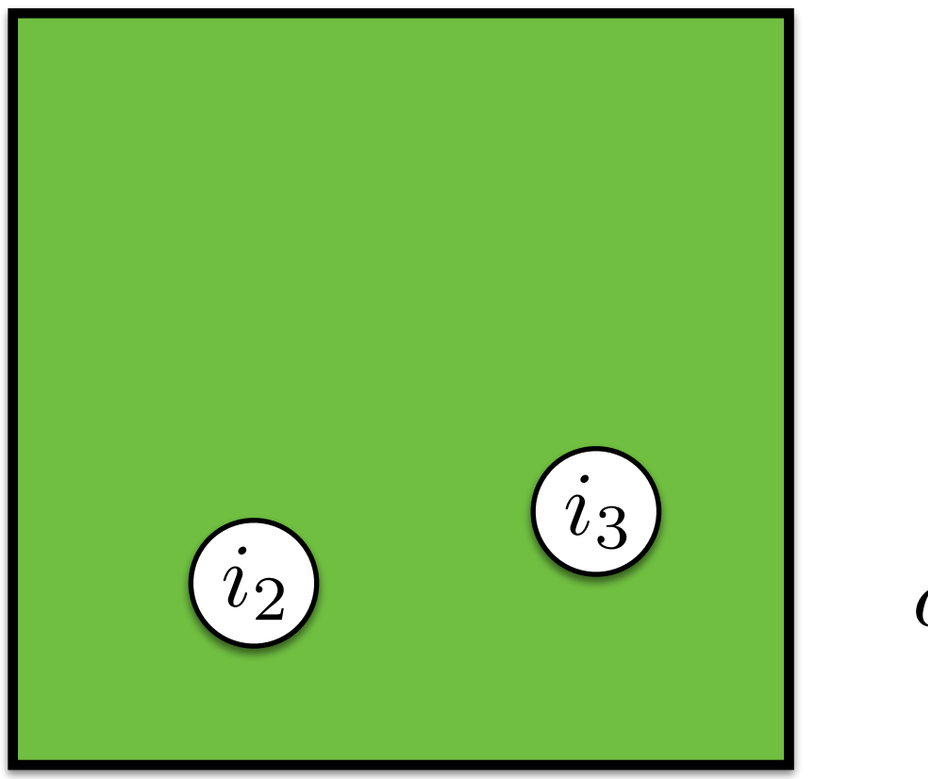
B

"maybe keep"

Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_3\}) - f(A)$ "gain"

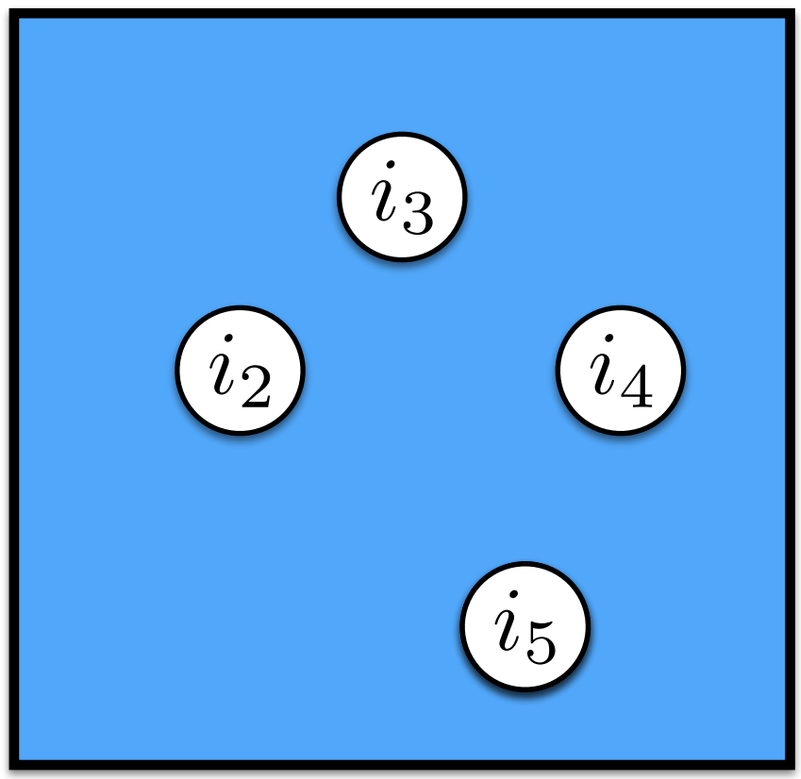
$\beta \leftarrow f(B \setminus \{i_3\}) - f(B)$ "harm"



A

"definitely keep"

$\alpha \geq \beta$?



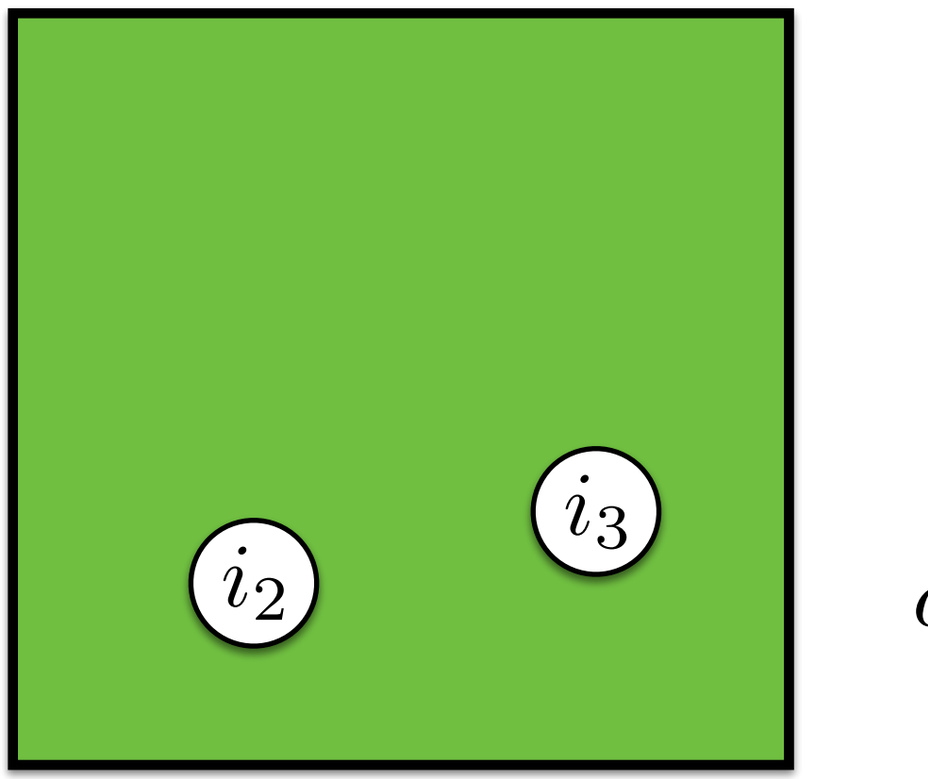
B

"maybe keep"

Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_3\}) - f(A)$ "gain"

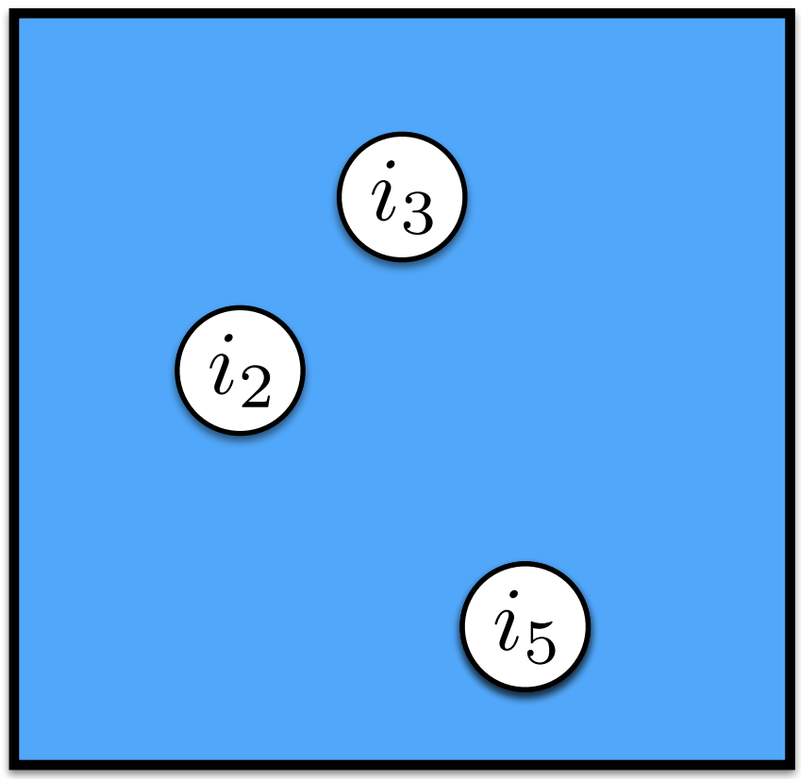
$\beta \leftarrow f(B \setminus \{i_3\}) - f(B)$ "harm"



A

"definitely keep"

$\alpha \geq \beta$?



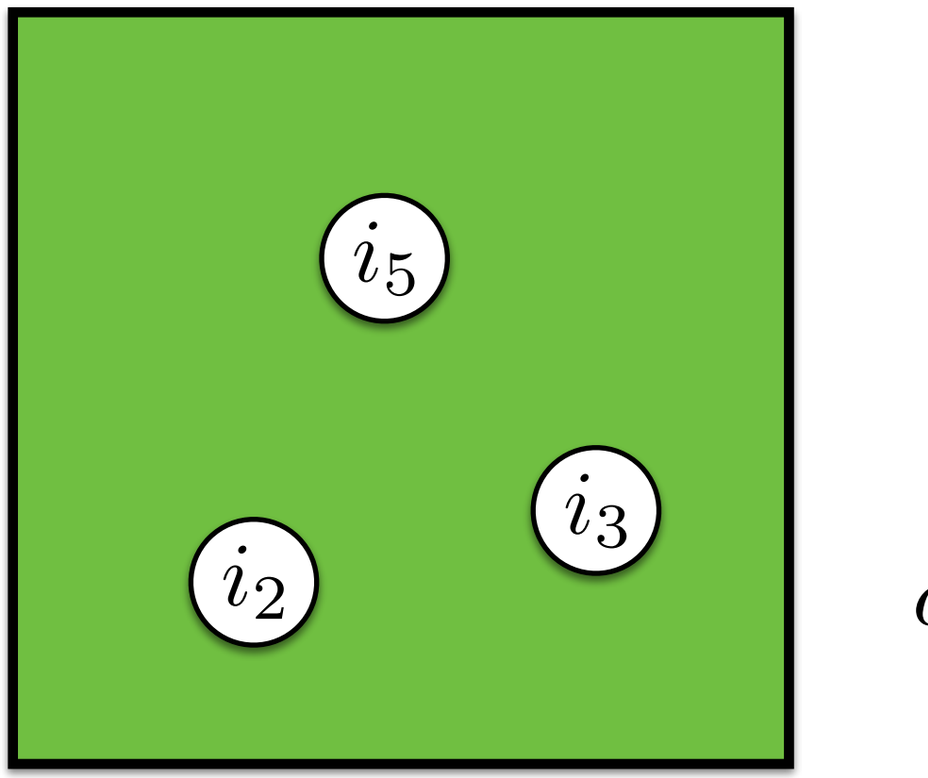
B

"maybe keep"

Deterministic double greedy

$\alpha \leftarrow f(A \cup \{i_3\}) - f(A)$ "gain"

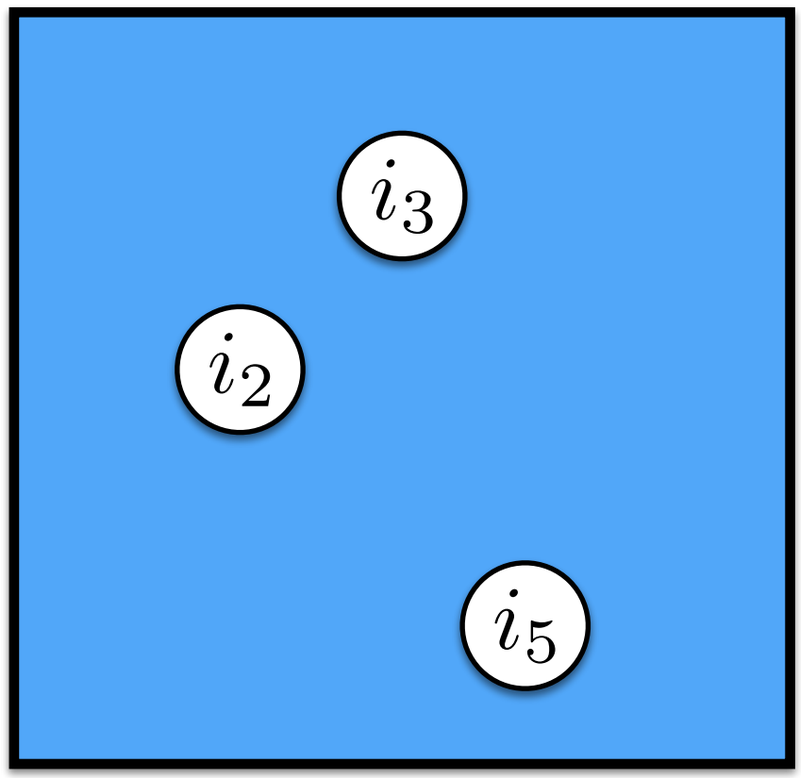
$\beta \leftarrow f(B \setminus \{i_3\}) - f(B)$ "harm"



A

"definitely keep"

$\alpha \geq \beta$?



B

"maybe keep"

Deterministic double greedy

$A \leftarrow \emptyset, B \leftarrow V$

Deterministic double greedy

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$\alpha \leftarrow f(A \cup \{i\}) - f(A)$ “gain”

$\beta \leftarrow f(B \setminus \{i\}) - f(B)$ “harm”

Deterministic double greedy

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- If $\alpha \geq \beta$, update *A*, else update *B*

Deterministic double greedy

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- If $\alpha \geq \beta$, update *A*, else update *B*

- Guarantee: 1/3-approximation (can we do better?)

Randomized double greedy

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- Update *A* w.p. $\alpha/(\alpha + \beta)$

- Guarantee: 1/2-approximation! (in expectation)

Hands-on Section

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- Add *i* to *A* w.p. $\alpha/(\alpha+\beta)$ else remove *i* from *B*

Hands-on Section

- Implement randomized double greedy

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- Add *i* to *A* w.p. $\alpha/(\alpha + \beta)$ else remove *i* from *B*

Hands-on Section

- Implement randomized double greedy
- Compare the three algorithms:

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

- Add *i* to *A* w.p. $\alpha/(\alpha + \beta)$ else remove *i* from *B*

Hands-on Section

- Implement randomized double greedy
- Compare the three algorithms:
  - Which performs best? Is it surprising?

$A \leftarrow \emptyset, B \leftarrow V$

- For each $i \in V$, compute

$$\alpha \leftarrow f(A \cup \{i\}) - f(A) \qquad \text{“gain”}$$

$$\beta \leftarrow f(B \setminus \{i\}) - f(B) \qquad \text{“harm”}$$

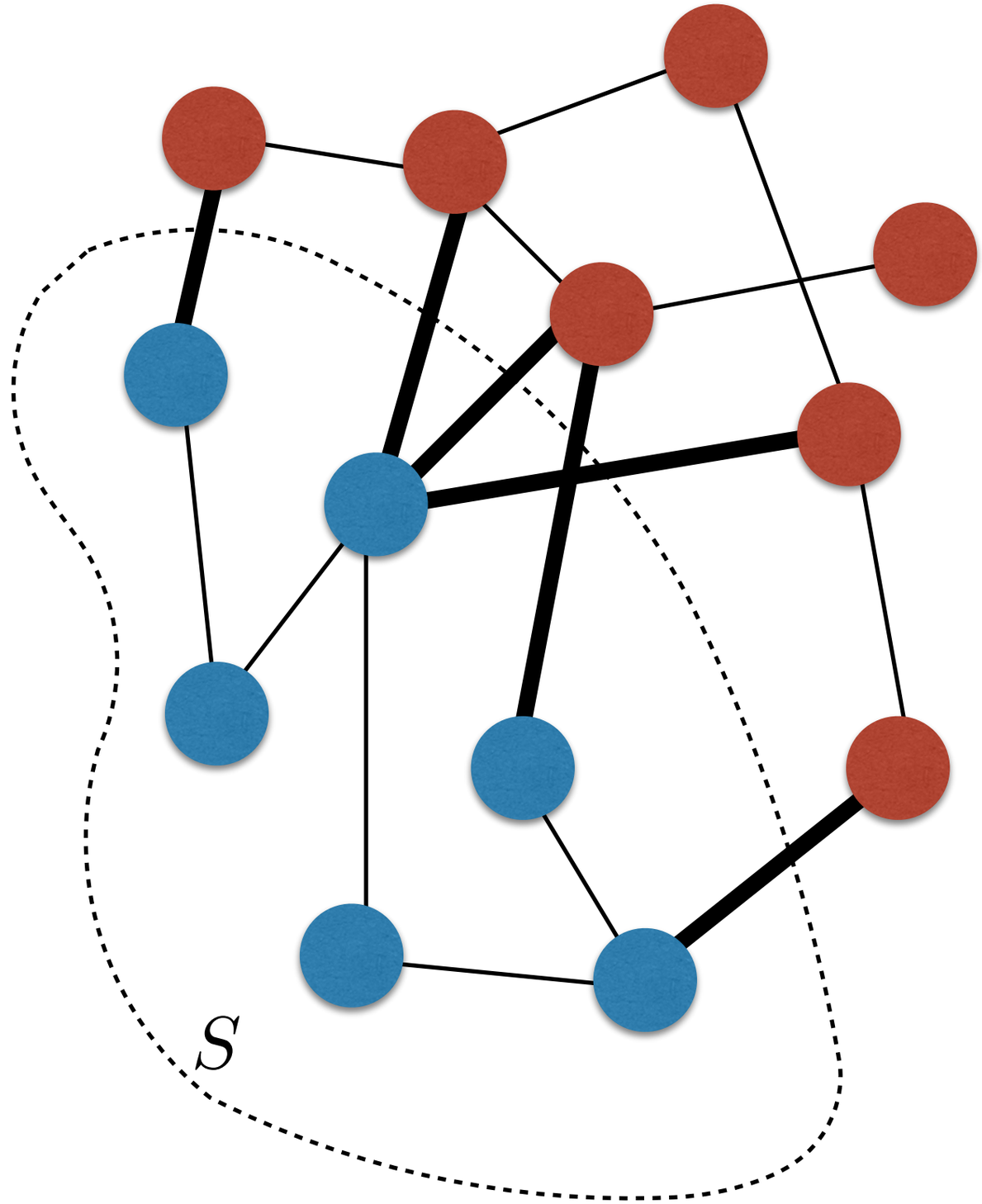
- Add *i* to *A* w.p. $\alpha/(\alpha+\beta)$ else remove *i* from *B*

How else to improve runtime?

- Strategies tried so far:
  - Different algorithms for different problems
  - Accelerated versions of existing algorithms
- Orthogonal strategy: make the marginal gains oracle more efficient

Cut functions

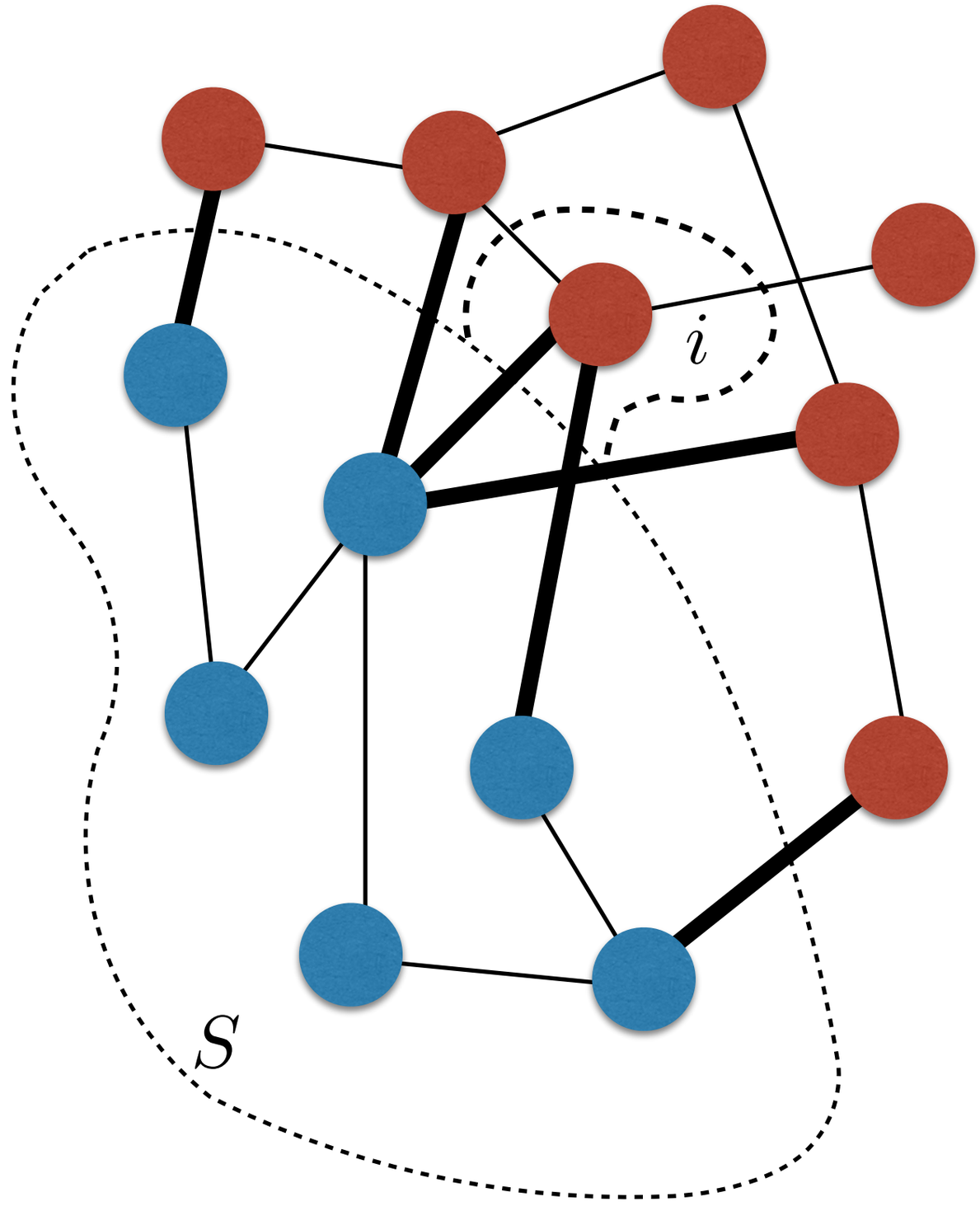
- Marginal gains function can be computed efficiently!



Cut functions

- Marginal gains function can be computed efficiently!

- When considering adding node *i*, need only look at its incident edges



Application: summarization

- Super large set *V* of items, e.g. images, documents
- Goal: find small subset *S* that can represent *V*
- Idea: define set function $f(S)$ capturing quality of *S*
- Then solve $\max_{S:|S|\leq k} f(S)$

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*
  - e.g. inner product, kernel similarity…

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*
  - e.g. inner product, kernel similarity…
- What makes a good summary *S* of *V*?

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*
  - e.g. inner product, kernel similarity…
- What makes a good summary *S* of *V*?

- Idea 1: each *i* similar on average to all elements of S:

$$f(S) = \sum_{i \in V} \sum_{j \in S} s(i,j)$$

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*
  - e.g. inner product, kernel similarity…
- What makes a good summary *S* of *V*?

- Idea 1: each *i* similar on average to all elements of S:

$$f(S) = \sum_{i \in V} \sum_{j \in S} s(i, j)$$

all points in dataset

Deciding on *f(S)*

- Suppose *s(i,j)* = similarity score between items *i, j.*
  - e.g. inner product, kernel similarity…
- What makes a good summary *S* of *V*?

- Idea 1: each *i* similar on average to all elements of S:

$$f(S) = \sum_{i \in V} \sum_{j \in S} s(i, j)$$

all points in dataset

points in the summary

Deciding on *f(S)*

- Idea 2: each *i* similar to at least one element of S:

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j)$$

Deciding on *f(S)*

- Idea 2: each *i* similar to at least one element of S:

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j)$$

all points in dataset

best **point in the summary**

Deciding on *f(S)*

- Idea 2: each *i* similar to at least one element of S:

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j)$$

all points in dataset

***best* point in the summary**

- Idea 3: same, but penalize *S* for having elements that are too similar

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j) - \frac{1}{|V|} \sum_{i \in S} \sum_{j \in S} s(i, j)$$

Deciding on *f(S)*

- Idea 2: each *i* similar to at least one element of S:

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j)$$

all points in dataset

***best* point in the summary**

- Idea 3: same, but penalize *S* for having elements that are too similar

$$f(S) = \sum_{i \in V} \max_{j \in S} s(i, j) - \frac{1}{|V|} \sum_{i \in S} \sum_{j \in S} s(i, j)$$

***within* the summary**

Deciding on *f(S)*

- Idea 4: determinants: $f(S) = \log\det K_S$

$$\log\det K_S \propto \mathrm{Vol}^2(\{v_i\}_{i\in S})$$

Deciding on *f(S)*

- Idea 4: determinants: $f(S) = \log\det K_S$
- Intuition: if item *i* associated with vector x_i:

$$\log\det K_S \propto \mathrm{Vol}^2(\{v_i\}_{i\in S})$$

Details to keep in mind

- All of these are submodular. Not all are monotone!

Details to keep in mind

- All of these are submodular. Not all are monotone!
  - (bonus section with a new algorithm for that)

Details to keep in mind

- All of these are submodular. Not all are monotone!
  - (bonus section with a new algorithm for that)

- How do the different summaries reflect the different objective functions?

Details to keep in mind

- All of these are submodular. Not all are monotone!
  - (bonus section with a new algorithm for that)

- How do the different summaries reflect the different objective functions?

- How does performance change when classes are imbalanced?

Details to keep in mind

- All of these are submodular. Not all are monotone!
  - (bonus section with a new algorithm for that)

- How do the different summaries reflect the different objective functions?

- How does performance change when classes are imbalanced?

- How might you speed up optimization when the dataset is large?