Approximation Algorithms

Learning Goals.

- Introduce Approximation Algorithms:
  - (PTAS) Polynomial Time Approximation Schemes:
    polynomial time algorithms with approximation guarantees
- Load Balancing (Greedy Balance).
- Load Balancing (Longest Processing Time First).
- Vertex Cover (Linear Programming plus Rounding).

Readings: Read Chapter 11 of text, can omit section 11.7.

Based on Chapter 11, Kleinberg and Tardos

Approximation Algorithms

Q. Suppose I need to solve an NP-hard problem. What should I do?
A. Theory says you're unlikely to find a poly-time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in poly-time.
- Solve arbitrary instances of the problem.

We consider sacrificing optimality, but attempt to preserve a guarantee.

Approximation Algorithms (cont.)

ρ-approximation algorithm.

- Guaranteed to run in poly-time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.
  Let C be the computed value and C^* be the optimal, then:
  For a maximization problem, $C \geq C^*/\rho$, with $\rho \geq 1$.
  For a minimization problem, $C \leq \rho\, C^*$, with $\rho \geq 1$.
  Some authors use $\rho := 1/\rho \leq 1$ for maximization problems.

Load Balancing

Input. m identical machines; n jobs, job j has processing time t_j.

- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let J(i) be the subset of jobs assigned to machine i. The load of machine i is $L_i = \Sigma_{j \in J(i)}\, t_j$.

Def. The makespan is the maximum load on any machine $L = \max_i L_i$.

Load balancing. Assign each job to a machine to minimize makespan.

Load Balancing: List Scheduling

List-scheduling algorithm.
- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

```
List-Scheduling(m, n, t_1,t_2,…,t_n) {
   for i = 1 to m {
      L_i ← 0        ← load on machine i
      J(i) ← φ       ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmin_k L_k         ← machine i has smallest load
      J(i) ← J(i) ∪ {j}        ← assign job j to machine i
      L_i ← L_i + t_j          ← update load of machine i
   }
   return J, max{L_i}
}
```

Implementation. O(n log n) using a priority queue.

Load Balancing: List Scheduling Analysis

Theorem. [Graham, 1966] Greedy algorithm is a 2-approximation.
- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L*.

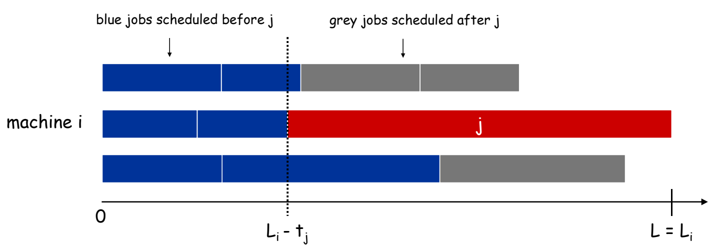
Lemma 1. The optimal makespan $L^* \geq \max_j t_j$.
Pf. Some machine must process the most time-consuming job. ▪

Lemma 2. The optimal makespan $L^* \geq \frac{1}{m} \sum_j t_j$.
Sketch of Pf.
- The total processing time is $\Sigma_j t_j$.
- One of m machines must do at least a 1/m fraction of total work. ▪

These provide useful lower bounds on the optimal makespan L*.

Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.
Pf. Consider load L_i of bottleneck machine i.
- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.



Load Balancing: List Scheduling Analysis

Theorem. Greedy algorithm is a 2-approximation.
Pf. Consider load L_i of bottleneck machine i.
- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m:

$$\begin{aligned} L_i - t_j &\leq \tfrac{1}{m}\sum_k L_k \\ &= \tfrac{1}{m}\sum_i t_i \\ \text{Lemma 2} \rightarrow\ &\leq L^* \end{aligned}$$

- Now $L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$. ▪

Lemma 1

Load Balancing: List Scheduling Analysis

Summary: For input s, let L(s) = L_i (the time required for the greedy schedule) and L*(s) be the minimum finish time. For ρ = 2 we have:

$\boxed{L(s)/L^*(s) \le \rho \text{ for all inputs } s.}$

That is, this greedy algorithm is a 2-approximation.

Remaining questions about this algorithm (all roughly similar in nature):

- Can L(s) actually be as bad as ρL*(s)?
- Is the estimate for ρ sharp?
- Does sup{L(s)/L*(s) | input s } = ρ (where sup denotes supremum)?
- For a given β in [1, ρ],

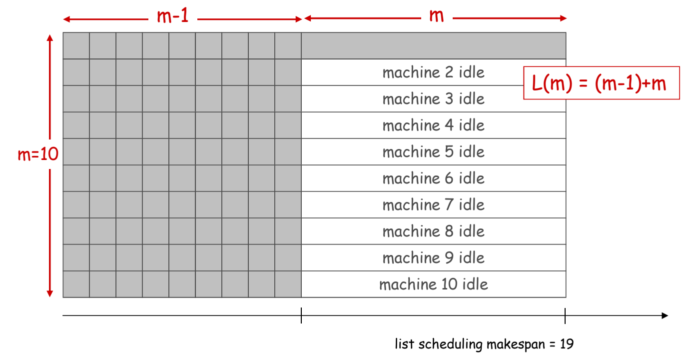
  $\boxed{\text{Does there exist an input } s \text{ such that } \beta \le L(s)/L^*(s)?}$

  E.g., β = 3/2 (we might try to certify this with one example input s.)
- For a given β in [1, ρ], is β ≤ sup{L(s)/L*(s) | input s }?

Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?
A. Essentially yes.

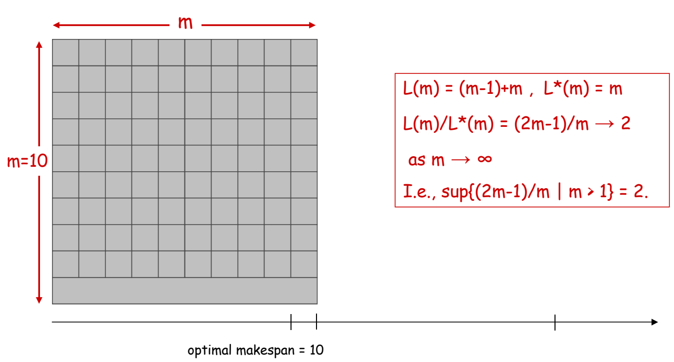
Ex: m machines, first m(m-1) jobs length 1 jobs, then one length m job



Load Balancing: List Scheduling Analysis

Q. Is our analysis tight?
A. Essentially yes.

Ex: m machines, m(m-1) jobs length 1 jobs, one job of length m



L(m) = (m-1)+m , L*(m) = m

L(m)/L*(m) = (2m-1)/m → 2 as m → ∞

I.e., sup{(2m-1)/m | m > 1} = 2.

Load Balancing: LPT Rule

Longest processing time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t1,t2,...,tn) {
   Sort jobs so that t1 ≥ t2 ≥ ... ≥ tn

   for i = 1 to m {
      Li ← 0             ← load on machine i
      J(i) ← ϕ           ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmink Lk         ← machine i has smallest load
      J(i) ← J(i) ∪ {j}      ← assign job j to machine i
      Li ← Li + tj           ← update load of machine i
   }
   return J, max{Li}
}
```

Load Balancing: LPT Rule

Observation. If at most m jobs, then list-scheduling is optimal.
Pf. Each job put on its own machine. ▪

Lemma 3. If there are more than m jobs, $L^* \geq 2\, t_{m+1}$.
Pf.
- Consider first m+1 jobs $t_1, \ldots, t_{m+1}$.
- Since the t_i's are in descending order, each takes at least t_{m+1} time.
- There are m+1 jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs. ▪

Theorem. LPT rule is a 3/2 approximation algorithm.
Pf. Same basic approach as for list scheduling.

$$L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq \frac{1}{2}L^*} \leq \tfrac{3}{2}L^*. \quad ▪$$

Lemma 3
(by observation, can assume number of jobs > m)

Load Balancing: LPT Rule

Q. Is our 3/2 analysis tight? I.e., is sup{L(s)/L*(s) | s } = 3/2?
A. No.

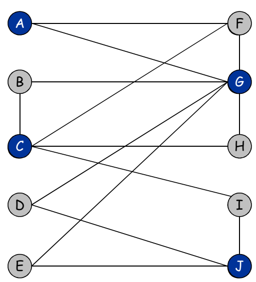
Theorem. [Graham, 1969] LPT rule is a 4/3-approximation.
Pf. More sophisticated analysis of same algorithm.

Q. Is Graham's 4/3 analysis tight?
A. Yes.

Ex: m machines, n = 2m jobs, with one job of length 3m, another job of length m, and then 2 jobs of each length m+1, m+2, …, 2m-1.

Approximating Vertex Cover

Vertex Cover. Given an undirected graph G = (V, E), find a minimum size subset of nodes S such that every edge is incident to at least one vertex in S.

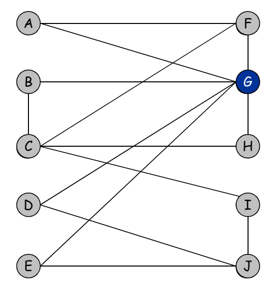


Minimum size vertex cover?

S* = { A, C, G, J}

Greedy by Degree Algorithm

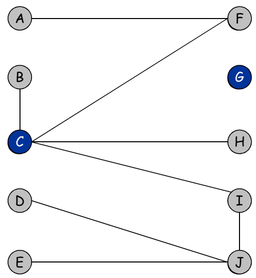
Greedy by Degree Algorithm for Vertex Cover: Select next vertex to be one with maximum degree. Remove it and its corresponding edges. Repeat until the remaining graph has no edges.



Step 1: S = { G }

Greedy by Degree Algorithm

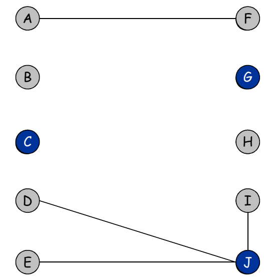
Greedy by Degree Algorithm for Vertex Cover: Select next vertex to be one with maximum degree. Remove it and its corresponding edges. Repeat until the remaining graph has no edges.



Step 2: S = { G, C }

Greedy by Degree Algorithm

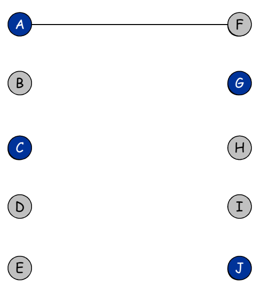
Greedy by Degree Algorithm for Vertex Cover: Select next vertex to be one with maximum degree. Remove it and its corresponding edges. Repeat until the remaining graph has no edges.



Step 3: S = { G, C, J }

Greedy by Degree Algorithm

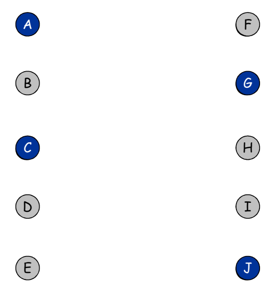
Greedy by Degree Algorithm for Vertex Cover: Select next vertex to be one with maximum degree. Remove it and its corresponding edges. Repeat until the remaining graph has no edges.



Step 4: S = { G, C, J, A }

Greedy by Degree Algorithm

Greedy by Degree Algorithm for Vertex Cover: Select next vertex to be one with maximum degree. Remove it and its corresponding edges. Repeat until the remaining graph has no edges.



Return: S = { G, C, J, A }

Same as S* = { A, C, G, J} for this example

Analysis of Greedy by Degree

Is the Greedy by Degree algorithm a ρ-approximation?

That is, for
- Input s describing a graph G = (V, E).
- L(s) defined to be |S|, where S is the output of greedy by degree.
- L*(S) defined to be |S*|, where S* is a minimum sized vertex cover.

Then, does there exist a constant ρ such that

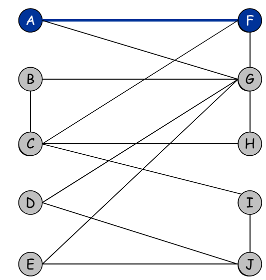
$\boxed{L(s)/L^*(s) \leq \rho \text{ for all inputs } s?}$

Answer: No. Blackboard: Graphs s=(V,E) s.t. $L(s)/L^*(s) \in \Theta(\log|V|)$.

Conclusion: The Greedy by Degree algorithm is not a ρ-approximation, for any constant ρ. Turns out it is a O(log(|V|))-approximation.

Edge Removal Algorithm

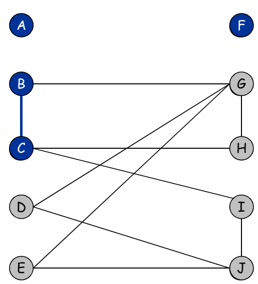
Edge Removal Algorithm for Vertex Cover: Select an edge. Add both endpoints to vertex cover. Remove all edges terminating at these endpoints . Repeat until the remaining graph has no edges.
(E.g. Select edges in lexical order: ((A,F), (A,G), (B,C), … (I, J))



Step 1: S = { A, F }

Edge Removal Algorithm

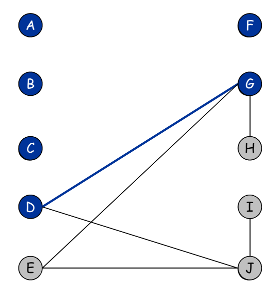
Edge Removal Algorithm for Vertex Cover: Select an edge. Add both endpoints to vertex cover. Remove all edges terminating at these endpoints . Repeat until the remaining graph has no edges.
(E.g. Select edges in lexical order: ((A,F), (A,G), (B,C), … (I, J))



Step 2: S = { A, F, B, C }

Edge Removal Algorithm

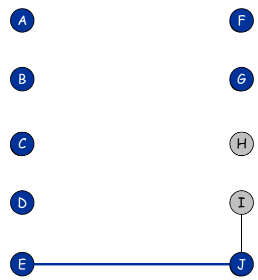
Edge Removal Algorithm for Vertex Cover: Select an edge. Add both endpoints to vertex cover. Remove all edges terminating at these endpoints . Repeat until the remaining graph has no edges.
(E.g. Select edges in lexical order: ((A,F), (A,G), (B,C), … (I, J))



Step 3: S = { A, F, B, C, D, G }

Edge Removal Algorithm

Edge Removal Algorithm for Vertex Cover: Select an edge. Add both endpoints to vertex cover. Remove all edges terminating at these endpoints . Repeat until the remaining graph has no edges.
(E.g. Select edges in lexical order: ((A,F), (A,G), (B,C), … (I, J))

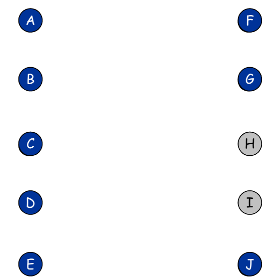


Step 4: S = { A, F, B, C, D, G, E, J }

Edge Removal Algorithm

Edge Removal Algorithm for Vertex Cover: Select an edge. Add both endpoints to vertex cover. Remove all edges terminating at these endpoints . Repeat until the remaining graph has no edges.
(E.g. Select edges in lexical order: ((A,F), (A,G), (B,C), … (I, J))

Apply this alg to the examples used to show Greedy by Degree is not a ρ-approximation



Return: S = { A, F, B, C, D, G, E, J }, |S| = 8.

Recall S* = { A, C, G, J} for this example, |S*| = 4.

Edge Removal is a 2-approximation

Theorem. If S* is a vertex cover of minimum size, then any S produced by the Edge Removal Alg. is a vertex cover with |S| ≤ 2|S*|.

Pf. [S is a vertex cover]

- In the alg, an edge (u, v) ∈ E is removed only when covered.
- Algorithm terminates when there are no edges left, with all edges covered.

Pf. [|S | is at most size 2 |S*|]

- Let S* be optimal vertex cover.
- Let E_S be the set of edges selected by Edge Removal (where both endpoints are added to S).
- For every e ∈ E_S, then e must have at least one endpoint in S*:

$$|S| = \sum_{e \in E_S} 2 = 2|E_S| \leq 2|S^*|$$

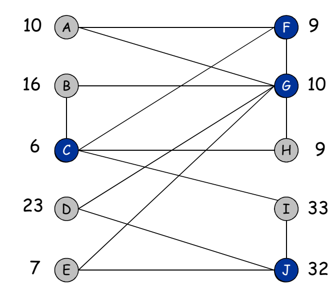
Both endpoints of e ∈ E_S in S

Each e ∈ E_S has at least one endpoint in S*

Moral: Less greed is sometimes better (e.g., this vs greedy-by-degree).

Weighted Vertex Cover

Weighted vertex cover. Given an undirected graph G = (V, E) with vertex i having weight $w_i \geq 0$, find a minimum weight subset of nodes S such that every edge is incident to at least one vertex in S.



Total weight (9 + 10 + 6 + 32)= 57. Minimum?

Weighted Vertex Cover: ILP Formulation

Weighted vertex cover. Given an undirected graph $G = (V, E)$ with vertex i having weight $w_i \geq 0$, find a minimum weight subset of nodes S such that every edge is incident to at least one vertex in S.

Integer linear programming formulation.

- Model inclusion of each vertex i using a 0/1 variable x_i.

$$x_i = \begin{cases} 0 & \text{if vertex } i \text{ is not in vertex cover} \\ 1 & \text{if vertex } i \text{ is in vertex cover} \end{cases}$$

  Vertex covers in 1-1 correspondence with 0/1 assignments: $S = \{i \in V : x_i = 1\}$

- Objective function: minimize $\Sigma_i w_i x_i$.
- For each edge $(i,j) \in E$, must take either i or j: $x_i + x_j \geq 1$.

Weighted Vertex Cover: ILP Formulation

Weighted vertex cover. Integer linear programming formulation.

$$\begin{array}{llccl} (ILP) \ \min & \sum_{i \in V} w_i x_i & & & \\ \text{s. t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\ & x_i & \in & \{0,1\} & i \in V \end{array}$$

Observation. If x^* is optimal solution to (ILP), then $S = \{i \in V : x^*_i = 1\}$ is a min weight vertex cover.

Weighted Vertex Cover: Real-Valued Relaxation

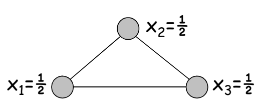
Weighted vertex cover. Linear programming formulation.

$$\begin{array}{llccl} (LP) \ \min & \sum_{i \in V} w_i x_i & & & \\ \text{s. t.} & x_i + x_j & \geq & 1 & (i,j) \in E \\ & x_i & \geq & 0 & i \in V \end{array}$$

Note, $x_i \leq 1$ is not necessary.

Observation. Optimal value of (LP) is $\leq$ optimal value of (ILP).
Pf. LP has fewer constraints.

Note. LP is not equivalent to vertex cover.



weights w_i = 1 for each vertex
min-ILP: W = 2
min-LP: W = 1.5 (result shown)
rounded result from min-LP: W = 3

Q. How can solving LP help us find a small vertex cover?
A. Solve LP and round fractional values.

LP + Rounding is a 2-Approx for Weighted Vertex Cover

Theorem. If x^* is optimal solution to (LP), then $S = \{i \in V : x^*_i \geq \frac{1}{2}\}$ is a vertex cover whose weight W_{RLP} is at most twice the min possible weight W^*.

Pf. [S is a vertex cover]

- Consider an edge $(i, j) \in E$.
- Since $x^*_i + x^*_j \geq 1$, either $x^*_i \geq \frac{1}{2}$ or $x^*_j \geq \frac{1}{2}$ $\Rightarrow$ (i, j) covered.

Pf. [S has desired cost]

- Let S^* be optimal vertex cover. Then

$$W^* = \sum_{i \in S^*} w_i \underbrace{\geq}_{\text{LP is a relaxation}} \sum_{i \in V} w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \underbrace{\geq}_{x_i^* \geq \frac{1}{2}} \tfrac{1}{2} \sum_{i \in S} w_i = \tfrac{1}{2} W_{RLP}$$

- Therefore $W_{RLP} \leq 2W^*$ and the rounded LP algorithm is a 2-approximation.

Weighted Vertex Cover

Theorem (above). There exists a 2-approximation algorithm for weighted vertex cover.

Theorem. [Dinur-Safra 2001] If P ≠ NP, then no ρ-approximation for $\rho < 1.3607$, even with unit weights.

$10\sqrt{5} - 21$

Open research problem. Close the gap.

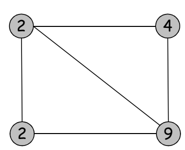
Extra Slides

Pricing Method for Weighted Vertex Cover

Pricing method. Each edge must be covered by some vertex i. Edge e pays price $p_e \geq 0$ to use vertex i.

Fairness. Edges incident to vertex i should pay $\leq w_i$ in total.

$$\text{for each vertex } i: \sum_{e=(i,j)} p_e \leq w_i$$



Claim. For any vertex cover S and any fair prices p_e: $\sum_e p_e \leq w(S)$.

Proof. ▪

$$\underbrace{\sum_{e \in E} p_e \leq}_{\text{each edge e covered by at least one node in S}} \sum_{i \in S} \; \underbrace{\sum_{e=(i,j)} p_e \leq}_{\text{sum fairness inequalities for each node in S}} \sum_{i \in S} w_i = w(S).$$

Pricing Method

Pricing method. Set prices and find vertex cover simultaneously.

```
Weighted-Vertex-Cover-Approx(G, w) {
   foreach e in E
      pe = 0

   while (∃ edge i-j such that neither i nor j are tight)
      select such an edge e
      increase pe without violating fairness
   }

   S ← set of all tight nodes
   return S
}
```

$$\sum_{e=(i,j)} p_e = w_i$$

Pricing Method

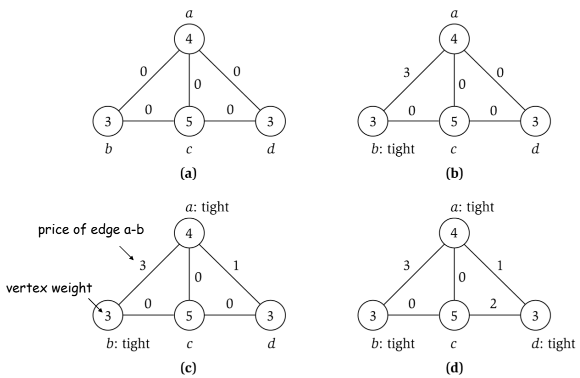


Figure 11.8

Pricing Method: Analysis

Theorem. Pricing method is a 2-approximation.

Pf.

- Algorithm terminates since at least one new node becomes tight after each iteration of while loop.
- Let S = set of all tight nodes upon termination of algorithm. S is a vertex cover: if some edge i-j is uncovered, then neither i nor j is tight. But then while loop would not terminate.
- Let S* be optimal vertex cover. We show w(S) ≤ 2w(S*).

$$w(S) = \sum_{i \in S} w_i \underset{\text{all nodes in S are tight}}{=} \sum_{i \in S} \sum_{e=(i,j)} p_e \underset{\substack{S \subseteq V,\\ \text{prices} \geq 0}}{\leq} \sum_{i \in V} \sum_{e=(i,j)} p_e \underset{\text{each edge counted twice}}{=} 2 \sum_{e \in E} p_e \underset{\text{fairness lemma}}{\leq} 2w(S^*). \quad \blacksquare$$