Approximation Algorithms

Ron Reaser

Lane Department of Computer Science and Electrical Engineering
West Virginia University

March 31, 2009

Approximability

For **NP**-complete problems, we want polytime means to find "good enough" solutions that are not too far from the optimal. This is called *approximation*.

Types of Problems

We may be able to develop approximation algorithms for optimization and search problems, but we cannot for decision problems: "yes"/"no" answers cannot be approximated in any meaningful manner. We do not need approximation algorithms for problems that can already be solved in polytime, because these would not be useful.

Approximability

For **NP**-complete problems, we want polytime means to find "good enough" solutions that are not too far from the optimal. This is called *approximation*.

Types of Problems

We may be able to develop approximation algorithms for optimization and search problems, but we cannot for decision problems: "yes"/"no" answers cannot be approximated in any meaningful manner. We do not need approximation algorithms for problems that can already be solved in polytime, because these would not be useful.

Approximability

For **NP**-complete problems, we want polytime means to find "good enough" solutions that are not too far from the optimal. This is called *approximation*.

Types of Problems

We may be able to develop approximation algorithms for optimization and search problems, but we cannot for decision problems: "yes"/"no" answers cannot be approximated in any meaningful manner. We do not need approximation algorithms for problems that can already be solved in polytime, because these would not be useful.

Approximability

For **NP**-complete problems, we want polytime means to find "good enough" solutions that are not too far from the optimal. This is called *approximation*.

Types of Problems

We may be able to develop approximation algorithms for optimization and search problems, but we cannot for decision problems: "yes" / "no" answers cannot be approximated in any meaningful manner. We do not need approximation algorithms for problems that can already be solved in polytime, because these would not be useful.

Approximability

For **NP**-complete problems, we want polytime means to find "good enough" solutions that are not too far from the optimal. This is called *approximation*.

Types of Problems

We may be able to develop approximation algorithms for optimization and search problems, but we cannot for decision problems: "yes" / "no" answers cannot be approximated in any meaningful manner. We do not need approximation algorithms for problems that can already be solved in polytime, because these would not be useful.

1. Introduction
   - Notion of Approximability
   - Constant Factor Approximations

2. Problems
   - Node Cover
   - Independent Set
   - MaxSat
   - Max-Cut
   - TSP
   - Knapsack

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Minimization Problems

Let M_1 be a minimization problem and x be any valid input. Let A_1 be a polytime algorithm for which $A_1(x)$ is a feasible solution iff $M_1(x)$ is a feasible solution. Then A_1 is an ϵ-approximation algorithm if

$$A_1(x) \leq \epsilon \cdot M_1(x)$$

for some constant $\epsilon \geq 1$.

Maximization Problems

Let M_2 be a maximization problem and x be any valid input. Let A_2 be a polytime algorithm for which $A_2(x)$ is a feasible solution iff $M_2(x)$ is a feasible solution. Then A_2 is an ϵ-approximation algorithm if

$$A_2(x) \geq \epsilon \cdot M_2(x)$$

for some constant $0 \leq \epsilon \leq 1$.

Summary

An *ϵ-approximation algorithm* will find solutions within at worst the constant factor ϵ times the optimum for all inputs of a given optimization problem.

Observation

There could only be a polytime 1-approximation algorithm for any **NP**-complete optimization problem if **P** = **NP**.

Approximation Threshold

The *approximation threshold* for a problem is the best known ϵ for which there is an ϵ-approximation algorithm for that problem.

Reductions

Reductions from one **NP**-complete problem to another tend not to preserve approximation thresholds. This will be seen later with INDEPENDENT SET.

Observation

There could only be a polytime 1-approximation algorithm for any **NP**-complete optimization problem if **P** = **NP**.

Approximation Threshold

The *approximation threshold* for a problem is the best known ϵ for which there is an ϵ-approximation algorithm for that problem.

Reductions

Reductions from one **NP**-complete problem to another tend not to preserve approximation thresholds. This will be seen later with INDEPENDENT SET.

Observation

There could only be a polytime 1-approximation algorithm for any **NP**-complete optimization problem if **P** = **NP**.

Approximation Threshold

The *approximation threshold* for a problem is the best known ϵ for which there is an ϵ-approximation algorithm for that problem.

Reductions

Reductions from one **NP**-complete problem to another tend not to preserve approximation thresholds. This will be seen later with INDEPENDENT SET.

Observation

There could only be a polytime 1-approximation algorithm for any **NP**-complete optimization problem if $\mathbf{P} = \mathbf{NP}$.

Approximation Threshold

The *approximation threshold* for a problem is the best known ϵ for which there is an ϵ-approximation algorithm for that problem.

Reductions

Reductions from one **NP**-complete problem to another tend not to preserve approximation thresholds. This will be seen later with INDEPENDENT SET.

Node-Cover

For a graph $G = (V, E)$, find the smallest set of nodes $C \subseteq V$ such that every edge in E has at least one of its nodes in C.

Goal

Find a good minimizing heuristic for this **NP**-complete problem.

First Approach

Consider that if a node has a high degree it is probably useful for covering many edges. This leads to a greedy heuristic as follows: starting with $C = \emptyset$, while there are still edges left in G find a node of highest degree, add it to C, and remove it from G.

Unfortunately this is not an ϵ-approximation algorithm for Node-Cover because it is off by a factor that is logarithmic in n rather than constant.

Node-Cover

For a graph $G = (V, E)$, find the smallest set of nodes $C \subseteq V$ such that every edge in E has at least one of its nodes in C.

Goal

Find a good minimizing heuristic for this **NP**-complete problem.

First Approach

Consider that if a node has a high degree it is probably useful for covering many edges. This leads to a greedy heuristic as follows: starting with $C = \emptyset$, while there are still edges left in G find a node of highest degree, add it to C, and remove it from G.

Unfortunately this is not an ϵ-approximation algorithm for Node-Cover because it is off by a factor that is logarithmic in n rather than constant.

Node-Cover

For a graph $G = (V, E)$, find the smallest set of nodes $C \subseteq V$ such that every edge in E has at least one of its nodes in C.

Goal

Find a good minimizing heuristic for this **NP**-complete problem.

First Approach

Consider that if a node has a high degree it is probably useful for covering many edges. This leads to a greedy heuristic as follows: starting with $C = \emptyset$, while there are still edges left in G find a node of highest degree, add it to C, and remove it from G.

Unfortunately this is not an ϵ-approximation algorithm for Node-Cover because it is off by a factor that is logarithmic in n rather than constant.

Node-Cover

For a graph $G = (V, E)$, find the smallest set of nodes $C \subseteq V$ such that every edge in E has at least one of its nodes in C.

Goal

Find a good minimizing heuristic for this **NP**-complete problem.

First Approach

Consider that if a node has a high degree it is probably useful for covering many edges. This leads to a greedy heuristic as follows: starting with $C = \emptyset$, while there are still edges left in G find a node of highest degree, add it to C, and remove it from G.

Unfortunately this is not an ϵ-approximation algorithm for Node-Cover because it is off by a factor that is logarithmic in n rather than constant.

Node-Cover

For a graph $G = (V, E)$, find the smallest set of nodes $C \subseteq V$ such that every edge in E has at least one of its nodes in C.

Goal

Find a good minimizing heuristic for this **NP**-complete problem.

First Approach

Consider that if a node has a high degree it is probably useful for covering many edges. This leads to a greedy heuristic as follows: starting with $C = \emptyset$, while there are still edges left in G find a node of highest degree, add it to C, and remove it from G.

Unfortunately this is not an ϵ-approximation algorithm for Node-Cover because it is off by a factor that is logarithmic in n rather than constant.

Second Approach

Trying to be simpler, consider the following heuristic: starting with $C = \emptyset$, while there are still edges left in G choose arbitrarily any one, add its endnodes to C, and delete it from G.

The nodes of C will comprise a matching of $\frac{1}{2}|C|$ edges in G. Any node cover, even the optimum, must have at least one node from each edge in a matching of G. So this is a 2-approximation algorithm for NODE-COVER.

Because this is the best known ϵ, the approximation threshold is 2 also.

Second Approach

Trying to be simpler, consider the following heuristic: starting with $C = \emptyset$, while there are still edges left in G choose arbitrarily any one, add its endnodes to C, and delete it from G.

The nodes of C will comprise a matching of $\frac{1}{2}|C|$ edges in G. Any node cover, even the optimum, must have at least one node from each edge in a matching of G. So this is a 2-approximation algorithm for NODE-COVER.

Because this is the best known ϵ, the approximation threshold is 2 also.

Second Approach

Trying to be simpler, consider the following heuristic: starting with $C = \emptyset$, while there are still edges left in G choose arbitrarily any one, add its endnodes to C, and delete it from G.

The nodes of C will comprise a matching of $\frac{1}{2}|C|$ edges in G. Any node cover, even the optimum, must have at least one node from each edge in a matching of G. So this is a 2-approximation algorithm for NODE-COVER.

Because this is the best known ϵ, the approximation threshold is 2 also.

Second Approach

Trying to be simpler, consider the following heuristic: starting with $C = \emptyset$, while there are still edges left in G choose arbitrarily any one, add its endnodes to C, and delete it from G.

The nodes of C will comprise a matching of $\frac{1}{2}|C|$ edges in G. Any node cover, even the optimum, must have at least one node from each edge in a matching of G. So this is a 2-approximation algorithm for NODE-COVER.

Because this is the best known ϵ, the approximation threshold is 2 also.

Second Approach

Trying to be simpler, consider the following heuristic: starting with $C = \emptyset$, while there are still edges left in G choose arbitrarily any one, add its endnodes to C, and delete it from G.

The nodes of C will comprise a matching of $\frac{1}{2}|C|$ edges in G. Any node cover, even the optimum, must have at least one node from each edge in a matching of G. So this is a 2-approximation algorithm for NODE-COVER.

Because this is the best known ϵ, the approximation threshold is 2 also.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $O(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless **P** = **NP**. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless **P** = **NP**. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless **P** $=$ **NP**. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless **P** = **NP**. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

Independent Set

An independent set in a graph is an induced set of nodes among which there are no edges. Given a graph $G = (V, E)$ where $n = |V|$, find the independent set I with the most nodes.

Observation

There is a simple reduction from Independent Set to Node Cover. (The maximum clique in $\bar{G}$ is exactly I in G.) However, despite the polytime approximability of Node Cover, there is no ϵ-approximation for Independent Set in time $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$ unless $\mathbf{P} = \mathbf{NP}$. In other words, it is not possible to distinguish in polytime whether $|I|$ is near 1 or near n. The proof is omitted herein.

Best Approach

The best known approximation is the following trivial heuristic: pick a vertex v from V and return $I = \{v\}$. Since $|I|$ is between 1 and n, this heuristic is an obviously polytime $\frac{1}{n}$-approximation. As the graph grows, $\frac{1}{n}$ approaches 0. Given that this is the best known approach, the approximation threshold is a useless 0.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

MaxSat

For a CNF formula of k-variable clauses $\phi_1, \phi_2, \ldots, \phi_n$ over variables $x_1, x_2, \ldots, x_m$, find a truth assignment for x that will satisfy the most ϕ.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider this very simple randomized heuristic: set each x to either **true** or **false** uniformly at random.

Each ϕ is expected to be satisfied with a probability of $\frac{2^k-1}{2^k}$ because there will be only one assignment, all-**false**, that will fail to satisfy the clause. In the worst case this would be for $k = 1$ where the expectation per ϕ would be $\frac{2^1-1}{2^1} = \frac{1}{2}$. By linearity of expectation, the sum of the expectations for all ϕ would be $\frac{1}{2}n$. So at least $\frac{n}{2}$ clauses are expected to be satisfied by this heuristic.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MaxSat.

Max-Cut

For a graph $G = (V, E)$, let a cut be a partitioning of the vertices in V such that some are in a set S and the rest are in $V - S$. A cut edge is an edge from a vertex in S to a vertex in $V - S$. The cut size is the number of cut edges. Maximize this quantity.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Max-Cut

For a graph $G = (V, E)$, let a cut be a partitioning of the vertices in V such that some are in a set S and the rest are in $V - S$. A cut edge is an edge from a vertex in S to a vertex in $V - S$. The cut size is the number of cut edges. Maximize this quantity.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Max-Cut

For a graph $G = (V, E)$, let a cut be a partitioning of the vertices in V such that some are in a set S and the rest are in $V - S$. A cut edge is an edge from a vertex in S to a vertex in $V - S$. The cut size is the number of cut edges. Maximize this quantity.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Max-Cut

For a graph $G = (V, E)$, let a cut be a partitioning of the vertices in V such that some are in a set S and the rest are in $V - S$. A cut edge is an edge from a vertex in S to a vertex in $V - S$. The cut size is the number of cut edges. Maximize this quantity.

Goal

Find a good maximizing heuristic for this **NP**-complete problem.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.

Randomized Approach

Consider a randomized heuristic similar to the previous one: for each $x \in V$, put x in S with probability $\frac{1}{2}$ uniformly at random.

For any edge $(i, j) \in E$, there are four possibilities.

(1) Node i is in S and node j is in S.

(2) Node i is in S and node j is in $V - S$, so (i, j) is a cut edge.

(3) Node i is in $V - S$ and node j is in S, so (i, j) is a cut edge.

(4) Node i is in $V - S$ and node j is in $V - S$.

For any edge, its expectation of being in the cut is therefore $\frac{1}{2}$. By linearity of expectation, the cut size expectation is the sum of the expectations for each cut, so the size of the cut cannot be worse than $\frac{|E|}{2}$.

Therefore this is a $\frac{1}{2}$-approximation algorithm for $\textsc{Max-Cut}$.

TSP

Given a graph $G = (V, E)$, find a tour (a cycle which visits every node in G exactly once or a Hamilton cycle) of minimum weight.

Observation

A polytime ϵ-approximation algorithm for this **NP**-complete problem can only exist if **P** = **NP**. If this were not so, the **NP**-complete problem HAMILTON-CYCLE could be decided in polytime.

Goal

Prove that a polytime ϵ-approximation algorithm does not exist for this problem.

TSP

Given a graph $G = (V, E)$, find a tour (a cycle which visits every node in G exactly once or a Hamilton cycle) of minimum weight.

Observation

A polytime ϵ-approximation algorithm for this **NP**-complete problem can only exist if **P** = **NP**. If this were not so, the **NP**-complete problem HAMILTON-CYCLE could be decided in polytime.

Goal

Prove that a polytime ϵ-approximation algorithm does not exist for this problem.

TSP

Given a graph $G = (V, E)$, find a tour (a cycle which visits every node in G exactly once or a Hamilton cycle) of minimum weight.

Observation

A polytime ϵ-approximation algorithm for this **NP**-complete problem can only exist if **P** = **NP**. If this were not so, the **NP**-complete problem HAMILTON-CYCLE could be decided in polytime.

Goal

Prove that a polytime ϵ-approximation algorithm does not exist for this problem.

TSP

Given a graph $G = (V, E)$, find a tour (a cycle which visits every node in G exactly once or a Hamilton cycle) of minimum weight.

Observation

A polytime ϵ-approximation algorithm for this **NP**-complete problem can only exist if **P** = **NP**. If this were not so, the **NP**-complete problem HAMILTON-CYCLE could be decided in polytime.

Goal

Prove that a polytime ϵ-approximation algorithm does not exist for this problem.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove $\mathbf{P} = \mathbf{NP}$. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove $\mathbf{P} = \mathbf{NP}$. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide Hamilton-Cycle in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove $\mathbf{P} = \mathbf{NP}$. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove **P** = **NP**. Win a prize and go home.

Proof

Given a graph $G = (V, E)$, construct a complete graph G' with all cities from V. The distance of edge $(i, j) \in G'$ is 1 if there is an edge $(i, j) \in G$ or ϵ otherwise. Now run the hypothetical polytime ϵ-approximation algorithm for TSP on graph G'. There are two possible outcomes.

(1) The returned tour has a cost of exactly $|V|$. This indicates a successful tour with $|V|$ edges of distance 1. This confirms the presence of a Hamilton cycle in G.

(2) The returned tour has a cost more than $|V|$ but no more than $\epsilon \cdot |V|$. This indicates the use of between 1 and $|V|$ edges of length ϵ in a tour of G'. This confirms the absence of a Hamilton cycle in G.

This would decide HAMILTON-CYCLE in polytime despite it being **NP**-complete.

This would prove $\mathbf{P} = \mathbf{NP}$. Win a prize and go home.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

KNAPSACK

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because KNAPSACK has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Knapsack

Given a set $I = \{1, 2, \ldots, n\}$ of items i with associated weights w_i and values v_i and given a weight limit W, find a subset $S \subseteq I$ such that $\sum_{i \in S} w_i \leq W$ with $\sum_{i \in S} v_i$ maximized. That is, find a selection of items such that their value is as high as possible while the sum of their weights does not exceed the weight limit W.

Observation

A polytime ϵ-approximation algorithm for this (not strongly) **NP**-complete maximization problem can be found for any $\epsilon < 1$. That is, solutions can be arbitrarily close to the optimum. Unlike with TSP, this does not prove that **P** = **NP** because Knapsack has a pseudopolynomial algorithm.

Goal

Prove that a polytime ϵ-approximation algorithm exists for this problem for any $\epsilon < 1$.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b\lfloor\frac{v_i}{2^b}\rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log\frac{\epsilon V}{n}\rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b\lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1})+w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$
\begin{aligned}
W(0,s) &= \infty \\
W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\}
\end{aligned}
$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i, s - v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{array}{lcl} W(0,s) & = & \infty \\ W(i+1,s) & = & \min\{W(i,s), W(i,s-v_{i+1})+w_{i+1}\} \end{array}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{array}{lcl} W(0,s) & = & \infty \\ W(i+1,s) & = & \min\{W(i,s), W(i, s - v_{i+1}) + w_{i+1}\} \end{array}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$
\begin{aligned}
W(0,s) &= \infty \\
W(i+1,s) &= \min\{W(i,s), W(i, s-v_{i+1}) + w_{i+1}\}
\end{aligned}
$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \leq i \leq n$ and $0 \leq s \leq n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \leq W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \le i \le n$ and $0 \le s \le n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i,s-v_{i+1})+w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \le W$. This solution is found in time $\mathcal{O}(n^2V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b\lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Proof

Let V be the maximum item value. Now we define the quantity $W(i,s)$ for $1 \le i \le n$ and $0 \le s \le n \cdot V$. This is the minimum weight attainable by selecting some of the first i items such that the sum of their values is exactly s. Recognize the following.

$$\begin{aligned} W(0,s) &= \infty \\ W(i+1,s) &= \min\{W(i,s), W(i, s - v_{i+1}) + w_{i+1}\} \end{aligned}$$

Pick the largest s such that $W(n,s) \le W$. This solution is found in time $\mathcal{O}(n^2 V)$ and is the standard pseudopolynomial algorithm for the optimal solution.

To speed it up, eschew accuracy in the v_i's if the numbers are very large. That is, redefine the values to be $u_i = 2^b \lfloor \frac{v_i}{2^b} \rfloor$ and use U as the maximum item value instead of V. This just means to ignore the b least significant bits.

This improves the running time to $\mathcal{O}(\frac{n^2 V}{2^b})$. The value returned for the solution item set is at most $n2^b$ less than the optimum. Because V is a lower bound on the value of the optimum solution, $\epsilon = \frac{n2^b}{V}$. This oddly allows for any $\epsilon < 1$ the truncation of $b = \lceil \log \frac{\epsilon V}{n} \rceil$ bits for the values to yield an $\mathcal{O}(\frac{n^3}{\epsilon})$ polytime algorithm.

This means KNAPSACK's approximation threshold is 1.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) = \approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Example

Where $b = 8$, consider the following items with their weights and values.

$i \in I$	w_i	v_i	$(v_i)_2$	$(u_i)_2$	u_i
1	5	9806	$10,0110,0100,1110$	$10,0110 * 2^8$	38
2	4	4570	$1,0001,1101,1010$	$1,0001 * 2^8$	17
3	5	1039	$100,0000,1111$	$100 * 2^8$	4
4	6	5413	$1,0101,0010,0101$	$1,0101 * 2^8$	21
5	4	6008	$1,0111,0111,1000$	$1,0111 * 2^8$	23
6	3	5014	$1,0011,1000,0110$	$1,0011 * 2^8$	19
7	3	2243	$1000,1100,0011$	$1000 * 2^8$	8

The algorithm can use $U = 38$ rather than $V = 9,806$. The approximate solution is off by a factor no more than $\frac{n2^b}{V} \approx 18\%$. The running time is $\mathcal{O}(\frac{n^2 V}{2^b}) \approx \mathcal{O}(1,877)$ instead of $\mathcal{O}(n^2 V) =\approx \mathcal{O}(480,494)$ which is 256 times as fast because many possible value sums are avoided during the test on $W(i,s)$.

Summary

Using a larger b will make the solution worse but the time better. Contrariwise, using a smaller b will make the solution better but the time worse. This is why the approximability is not limited.

Polytime Approximation Schemes

An algorithm which has such unlimited approximability is said to have a *polytime approximation scheme*. So KNAPSACK has a polytime approximation scheme.

Fully Polynomial Behavior

If a polynomial time ϵ-approximation algorithm depends polynomially in its time complexity on $\frac{1}{\epsilon}$, it is called *fully polynomial* as with KNAPSACK's time of $\mathcal{O}(\frac{n^3}{\epsilon})$.

Polytime Approximation Schemes

An algorithm which has such unlimited approximability is said to have a *polytime approximation scheme*. So KNAPSACK has a polytime approximation scheme.

Fully Polynomial Behavior

If a polynomial time ϵ-approximation algorithm depends polynomially in its time complexity on $\frac{1}{\epsilon}$, it is called *fully polynomial* as with KNAPSACK's time of $\mathcal{O}(\frac{n^3}{\epsilon})$.

Polytime Approximation Schemes

An algorithm which has such unlimited approximability is said to have a *polytime approximation scheme*. So KNAPSACK has a polytime approximation scheme.

Fully Polynomial Behavior

If a polynomial time ϵ-approximation algorithm depends polynomially in its time complexity on $\frac{1}{\epsilon}$, it is called *fully polynomial* as with KNAPSACK's time of $\mathcal{O}(\frac{n^3}{\epsilon})$.

Polytime Approximation Schemes

An algorithm which has such unlimited approximability is said to have a *polytime approximation scheme*. So KNAPSACK has a polytime approximation scheme.

Fully Polynomial Behavior

If a polynomial time ϵ-approximation algorithm depends polynomially in its time complexity on $\frac{1}{\epsilon}$, it is called *fully polynomial* as with KNAPSACK's time of $\mathcal{O}(\frac{n^3}{\epsilon})$.